

Minicurso de C++

Aula 04

Orientação a Objetos: Classes - Parte 2

Responsáveis:

Gabriel Daltro Duarte

Rafael Guerra de Pontes

Walisson da Silva

Wesley da Cunha Santos

Ianes Grécia



Separando a interface da implementação

Interface de uma classe

- Interfaces definem e padronizam o modo como as coisas, pessoas e sistemas interagem entre si (controles de rádio, por exemplo).
- Especificam as operações que um usuário pode realizar, mas não como essas operações estão implementadas.

Interface de uma classe

- A interface de uma classe descreve quais serviços os clientes de uma classe podem utilizar e como solicitá-los, mas não como a classe executa os serviços.

Divisão dos Arquivos

- Quando se cria uma classe, não é raro pretender que ela seja utilizada por outras pessoas posteriormente.
- Para “esconder” o código, separamos a definição e a implementação, utilizando, respectivamente, arquivos header (.h) e source (.cpp).

O Header (.h)

- Apenas aquilo que define como uma classe deve ser utilizada, basicamente seus atributos e os cabeçalhos de suas funções, sem qualquer referência a como as funções são implementadas.

Definições das funções-membro

- É realizada em um arquivo de código-fonte separado (.cpp).
- O identificador de cada função é precedido do pelo nome da classe e por ::, visto que, somente assim o compilador as reconhecerá como funções-membro da mesma classe.

Exemplo 01

```
#include <iostream>
#include <cmath>
using namespace std;
```

```
class Ponto {
private:
    double x, y;
public:
    Ponto(double cx, double cy){
        x = cx;
        y = cy;
    }
    Ponto(){
        x = y = 0;
    }
}
```

```
// Continuação
double distancia(){
    return sqrt((x*x) + (y*y));
}
```

```
};
```

```
int main(){
    Ponto p(3.0, 4.0);

    cout << p.distancia() << endl;

    return 0;
}
```


// Main

```
#include "Ponto.h"
```

```
using namespace std;
```

```
int main(){
```

```
    Ponto p(3.0, 4.0);
```

```
    cout << p.distancia()  
<< endl;
```

```
    return 0;
```

```
}
```

// Header

```
#include <iostream>
```

```
#include <cmath>
```

```
using namespace std;
```

```
class Ponto {
```

```
private:
```

```
    double x, y;
```

```
public:
```

```
    Ponto(double,
```

```
double);
```

```
    Ponto();
```

```
    double distancia();
```

```
};
```

// Funções-membro

```
#include "Ponto.h"
```

```
Ponto::Ponto(double cx, double  
cy){
```

```
    x = cx;
```

```
    y = cy;
```

```
}
```

```
Ponto::Ponto(){
```

```
    x = y = 0;
```

```
}
```

```
double Ponto::distancia(){
```

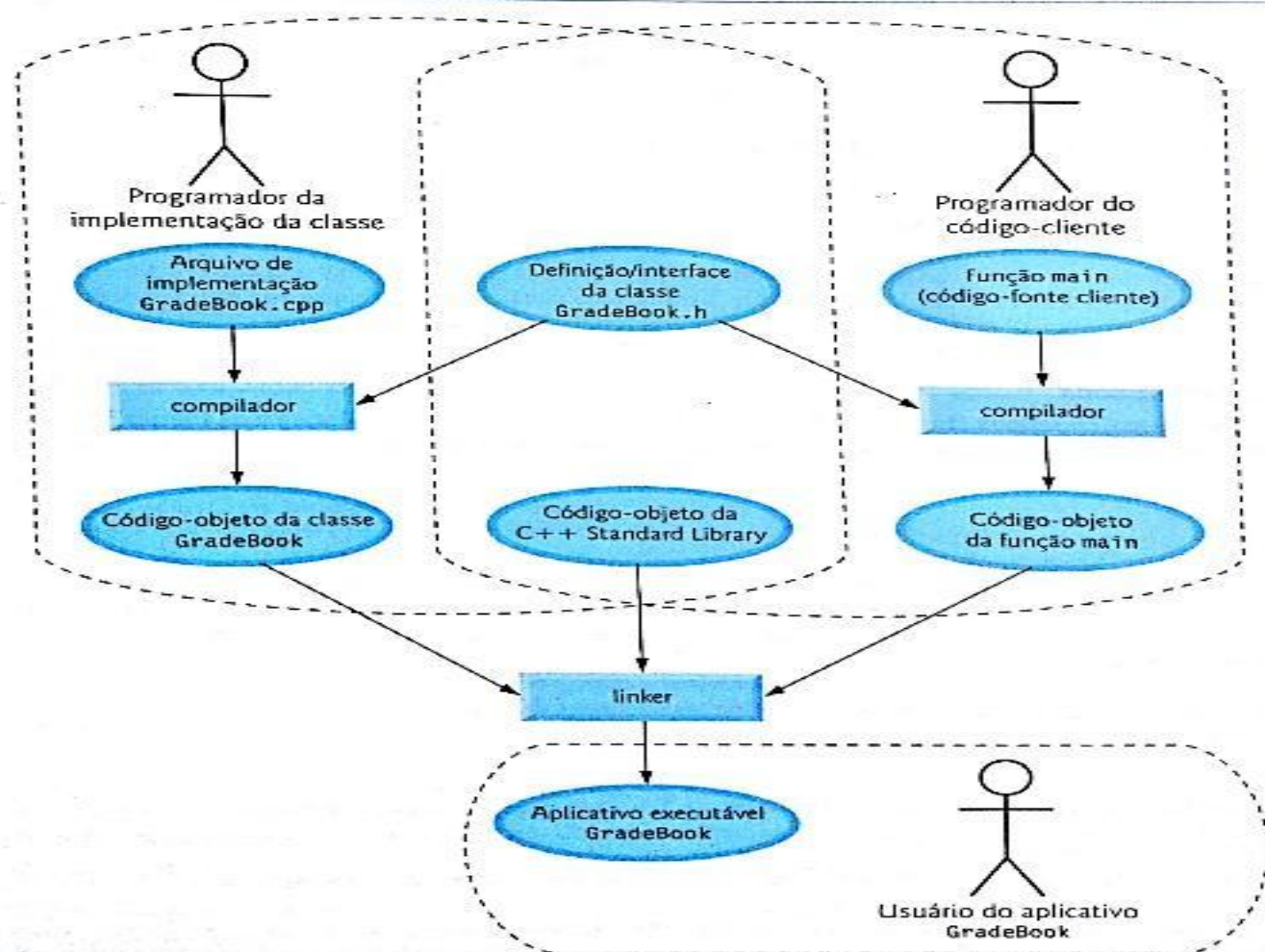
```
    return sqrt((x*x) +
```

```
(y*y));
```

```
}
```

O Processo de Compilação

- Como se dá o processo de compilação e vinculação dos arquivos de modo a “esconder” a implementação da classe?



Exercício

Crie uma classe Pessoa que contenha a altura e o sexo (M ou F) da pessoa, e uma função que calcule peso ideal, utilizando uma das seguintes fórmulas:

Para homens: $(72.7 * \text{altura}) - 58.0$

Para mulheres: $(62.1 * \text{altura}) - 44.7$

Em seguida separe a interface da implementação.

Herança

Herança

- Poderosa ferramenta na POO.
- Permite implementar uma classe a partir de outra classe.
- Facilita criação e manutenção através de reuso de código.
- Classe mais alta na hierarquia é chamada de classe base.
- Classes que herdam são chamadas classes derivadas.

Classes Base x Derivadas

- Uma classe pode derivar de mais de uma classe, podendo ter múltiplas classes-base.
- Utiliza-se uma lista de derivação para especificar as classes-base.
- Estrutura:

```
class ClasseDerivada: modificador_acesso ClasseBase1,  
modificador_acesso ClasseBase2, (...)
```

Controle de acesso e Herança

Acesso	private	protected	public
Própria classe	Sim	Sim	Sim
Classe filha (derivada)	Não	Sim	Sim
Classe externa	Não	Não	Sim


```
#include <iostream>

using namespace std;

class Pessoa{
protected:
    string nome;
    int idade;
public:
    Pessoa(string n = "", int i = 0): nome(n),
    idade(i){}
    string getNome(){
        return nome;
    }
    int getIdade(){
        return idade;
    }
};

class Funcionario: public Pessoa{
private:
    double salario;
```

```
public:
    Funcionario(Pessoa p, double s = 0){
        salario = s;
        nome = p.getNome();
        idade = p.getIdade();
    }

    void mostrarDados(){
        cout << "Dados: " << nome << ", ";
        cout << idade << " anos e ganha R$ ";
        cout << salario << ".\n";
    }
};

int main(){
    Pessoa p("Maria", 32);
    Funcionario f(p, 2521.52);
    f.mostrarDados();
    return 0;
}
```

Saída

C:\Users\Rafael\Desktop\MinicursoCPP\Aula04\heranca02.exe

Dados: Maria, 32 anos e ganha R\$ 2521.52.

```
#include <iostream>

using namespace std;

class Forma{
public:
    void setLargura(int l){
        largura = l;
    }
    void setAltura(int a){
        altura = a;
    }
protected:
    int largura;
    int altura;
};

class CustoTinta{
public:
    int getCusto(int area){
        return area * 15;
    }
};
```

```
class Retangulo: public Forma, public CustoTinta{
public:
    int getArea(){
        return (largura * altura);
    }
};

int main(void){
    Retangulo Ret;
    int area;

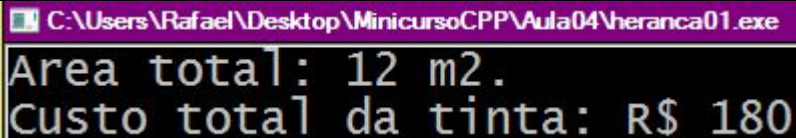
    Ret.setLargura(3);
    Ret.setAltura(4);

    area = Ret.getArea();

    cout << "Area total: " << Ret.getArea() << " m2." << endl;
    cout << "Custo total da tinta: R$ " << Ret.getCusto(area) << endl;

    return 0;
}
```

Saída



```
C:\Users\Rafael\Desktop\MinicursoCPP\Aula04\heranca01.exe
Area total: 12 m2.
Custo total da tinta: R$ 180
```

Exercício

Crie uma classe base `Animal` que possui atributos `protected` `nome`, `idade` e `peso`. Em seguida, crie uma classe derivada `Cachorro` que herde (derive) de `Animal`, possua um atributo `raça` e uma função que exiba todos os dados de `Cachorro`.

Polimorfismo

Polimorfismo

- Polimorfismo = qualidade ou estado de ser capaz de assumir diferentes formas.
- Ocorre quando há hierarquia de classes relacionadas por herança.
- Chamadas a funções-membro dependem do tipo do objeto que as invocam.

```
#include <iostream>
```

```
using namespace std;
```

```
class Animal{
```

```
protected:
```

```
    string nome;
```

```
public:
```

```
    Animal(string n){
```

```
        nome = n;
```

```
    }
```

```
    void falar(){
```

```
        cout << "Animal diz oi!\n";
```

```
    }
```

```
};
```

```
class Gato: public Animal{
```

```
public:
```

```
    Gato(string n): Animal(n){}
```

```
    void falar(){
```

```
        cout << "Gato diz oi... Miau!\n";
```

```
    }
```

```
};
```

```
class Galinha: public Animal{
```

```
public:
```

```
    Galinha(string n): Animal(n){}
```

```
    void falar(){
```

```
        cout << "Galinha diz oi... Cocoricooo!\n";
```

```
    }
```

```
};
```

```
int main(){
```

```
    Animal * pa;
```

```
    Gato gato("Frederico");
```

```
    Galinha galinha("Matilda");
```

```
    pa = &gato;
```

```
    pa->falar();
```

```
    pa = &galinha;
```

```
    pa->falar();
```

```
    return 0;
```

```
}
```

Qual o problema com esta saída?

C:\Users\Rafael\Desktop\MinicursoCPP\Aula04\polimorfismo01.exe

```
Animal diz oi!  
Animal diz oi!
```

Problema do exemplo anterior

- `void falar()` foi declarada na classe base e nas filhas, mas se ela não for virtual, temos uma resolução estática (***static resolution***).
- Também chamado de ***static linkage*** ou ***early binding***.
- O endereço de chamada da função `falar()` é ajustado durante a compilação.
- Qual a solução?
 - Declarar `falar()` como uma função virtual!

```
#include <iostream>
```

```
using namespace std;
```

```
class Animal{
```

```
protected:
```

```
    string nome;
```

```
public:
```

```
    Animal(string n){
```

```
        nome = n;
```

```
    }
```

```
    virtual void falar(){
```

```
        cout << "Animal diz oi!\n";
```

```
    }
```

```
};
```

```
class Gato: public Animal{
```

```
public:
```

```
    Gato(string n): Animal(n){}
```

```
    void falar(){
```

```
        cout << "Gato diz oi... Miau!\n";
```

```
    }
```

```
};
```

```
class Galinha: public Animal{
```

```
public:
```

```
    Galinha(string n): Animal(n){}
```

```
    void falar(){
```

```
        cout << "Galinha diz oi... Cocoricooo!\n";
```

```
    }
```

```
};
```

```
int main(){
```

```
    Animal * pa;
```

```
    Gato gato("Frederico");
```

```
    Galinha galinha("Matilda");
```

```
    pa = &gato;
```

```
    pa->falar();
```

```
    pa = &galinha;
```

```
    pa->falar();
```

```
    return 0;
```

```
}
```

Agora sim!

```
C:\Users\Rafael\Desktop\MinicursoCPP\Aula04\polimorfismo01.exe
```

```
Gato diz oi... Miau!
```

```
Galinha diz oi... Cocoricooo!
```


Explicação

- No exemplo anterior, o compilador observa o conteúdo (objeto) que está no endereço para o qual o ponteiro aponta antes de chamar a função.
- Assim, como Gato e Galinha têm a função falar() reimplementada, a chamada vai depender do objeto em ap.

Funções Virtuais

- É uma função em uma classe base declarada usando a palavra reservada `virtual`.
- O compilador não cria um link estático com funções virtuais.
- A chamada de funções virtuais durante a execução do programa vai depender do objeto que consta no endereço referenciado.
- Essa operação é chamada de ***dynamic linkage*** ou ***late binding***.

Classes Abstratas

(e funções virtuais puras)

Interface em C++

- Em C++, interfaces são **classes abstratas**.
- Uma classe é abstrata quando pelo menos uma de suas funções é **puramente virtual**.
- Para tanto, deve-se adicionar a palavra reservada “`virtual`” antes do tipo de retorno da função e “`= 0`” ao final de sua declaração.

Exemplo de Classe Abstrata

```
class FormaGeometrica{  
    public:  
        // Função virtual pura  
        virtual double getArea() = 0;  
    private:  
        double base;        // Base da forma  
        double altura;      // Altura da forma  
};
```

```

// Exemplo de Classe Abstrata
#include <iostream>

using namespace std;

// Base class
class Shape
{
public:
    // pure virtual function
    // interface framework.
    virtual int getArea() = 0;
    void setWidth(int w)
    {
        width = w;
    }
    void setHeight(int h)
    {
        height = h;
    }
protected:
    int width;
    int height;
};

// Derived classes
class Rectangle: public Shape
{
public:
    int getArea()
    {
        return (width * height);
    }
};

class Triangle: public Shape
{
public:
    int getArea()
    {
        return (width * height)/2;
    }
};

int main(void)
{
    Rectangle Rect;
    Triangle Tri;

    Rect.setWidth(5);
    Rect.setHeight(7);
    // Print the area of the object.
    cout << "Total Rectangle area: " <<
    Rect.getArea() << endl;

    Tri.setWidth(5);
    Tri.setHeight(7);
    // Print the area of the object.
    cout << "Total Triangle area: " <<
    Tri.getArea() << endl;

    return 0;
}

```

Fonte: http://www.tutorialspoint.com/cplusplus/cpp_interfaces.htm

Estratégia de Design

- Classes abstratas provêm uma interface comum apropriada para todas as aplicações externas.
- Herdar de uma classe abstrata padroniza o modo de operação de todas as classes derivadas.
- Novas classes podem ser posteriormente implementadas que se adequem ao molde estabelecido.

Alocação Dinâmica

Alocação Dinâmica de Memória

- Um bom entendimento de como a memória dinâmica funciona é essencial para se tornar um bom programador C++.
- Basicamente, a memória de um programa em C++ pode ser dividida em duas partes: *Stack* e *Heap*.

Alocação Dinâmica de Memória

- Muitas vezes não sabemos o quanto de memória iremos utilizar para salvar determinada informação em uma variável.
- Podemos alocar memória em tempo de execução dentro da heap utilizando um operador especial do C++ que retorna o endereço do espaço alocado.

O operador new

- Utilizamos o operador new para alocar memória dinamicamente para qualquer tipo de dado.
- Além de alocar memória, o operador new também constrói objetos, fator que o faz ser mais utilizado que a função `malloc()`, do C.

Alocacão dinâmica de memória

- Na memória *heap*, o usuário é diretamente responsável pela alocação e liberação do espaço utilizado.
- Para liberar uma variável alocada dinamicamente, utilizamos o operador `delete`.

Exemplo de uso do new e delete

```
#include <iostream>

using namespace std;

int main()
{
    int *p = NULL;
    p = new int;
    *p = 42;
    cout << "Valor de p: " << *p << endl;
    delete p;
    return 0;
}
```

Alocação dinâmica de arrays

- O operador **new** também serve para alocar arrays.

Exemplo: `int* ptr = new int [10];`

- A instrução acima aloca um array de 10 inteiros.
- O ponteiro `ptr` aponta o primeiro elemento do array.

Alocação dinâmica de arrays

- Observe que ao alocar dinamicamente um array de objetos não é possível inicializar cada objeto do array. Assim, cada objeto é inicializado por seu construtor padrão.
- Para desalocar um array alocado dinamicamente deve se utilizar a seguinte sintaxe:

```
delete [] ptr;
```

```
// Main
```

```
#include <iostream>
#include "ClassePonto.h"
int main()
{
    int N;
    std::cin >> N;
    Ponto* ptrPonto;
    ptrPonto = new Ponto [N];
    for (int i = 0; i < N; i++)
    {
        ptrPonto[i].imprimirPonto();
    }
    delete [] ptrPonto;
    return 0;
}
```

```
// Header
```

```
class Ponto {
public:
    Ponto(double = 0, double = 0);
    ~Ponto ()
    {
        std::cout << "Objeto destruido!\n";
    }
    void setX (double);
    void setY (double);
    void setXY (double, double);
    double getX (void) const;
    double getY (void) const;
    void imprimirPonto (void) const;
private:
    double x,y;
};
```

```
// Funções-membro
```

```
Ponto::Ponto (double a, double b)
{
    setX(a);
    setY(b);
    std::cout << "Objeto Ponto criado!\n";
}

...

void Ponto::imprimirPonto (void) const
{
    std::cout << "(" << getX() << "," << getY() <<
    ");\n";
}
```


Saída do exemplo do slide anterior

```
5
Objeto Ponto criado!
Objeto Ponto criado!
Objeto Ponto criado!
Objeto Ponto criado!
Objeto Ponto criado!
<0,0>;
<0,0>;
<0,0>;
<0,0>;
<0,0>;
Objeto destruido!
Objeto destruido!
Objeto destruido!
Objeto destruido!
Objeto destruido!
```

Exercício

Solicite ao usuário um inteiro n que corresponde à quantidade de elementos que serão armazenados em um array dinamicamente alocado com o operador `new`. Em seguida, leia n valores inteiros e os exiba na ordem inversa a que foram lidos. Não esqueça de liberar a memória com `delete`.

Construtores Padrão

Construtores Padrão

- **Construtor padrão** é um construtor que pode ser chamado sem ter que fornecer quaisquer argumentos, independentemente de o construtor ser auto-gerado ou user-definido. Só pode haver um construtor padrão por classe;
- Note-se que um construtor com parâmetros formais ainda pode ser chamado sem argumentos se argumentos padrão foram fornecidos na definição do construtor.

```
// Main
```

```
#include <iostream>
#include "ClassePonto.h"
```

```
using namespace std;
```

```
int main()
```

```
{
    Ponto P1;
    std::cout <<"P1: (" << P1.getX() << ","
    << P1.getY() << ")\n";

    return 0;
}
```

```
P1: <5.80271e-306,4.24399e-314>
```

```
// Ponto.h
```

```
class Ponto {
public:
    void setX (double);
    void setY (double);
    void setXY (double, double);
    double getX (void);
    double getY (void);
private:
    double x,y;
};
```

```
// Ponto.cpp
```

```
#include <iostream>
#include "ClassePonto.h"
```

```
double Ponto::getX (void)
{
    return x;
}
```

```
double Ponto::getY (void)
{
    return y;
}
```

```
void Ponto::setX (double a)
{
    x = a;
}
```

```
void Ponto::setY (double b)
{
    y = b;
}
```

```
•
•
•
```

```
// Main
```

```
#include <iostream>
#include "ClassePonto.h"
```

```
using namespace std;
```

```
int main()
{
    Ponto P1;
    std::cout << "P1: (" << P1.getX() << ", "
    << P1.getY() << ")\n";

    return 0;
}
```

```
Objeto Ponto criado!
P1: (2.81463e-307,4.24399e-314)
```

```
// Ponto.h
```

```
class Ponto {
public:
    Ponto ();
    void setX (double);
    void setY (double);
    void setXY (double, double);
    double getX (void);
    double getY (void);
private:
    double x,y;
};
```

```
// Ponto.cpp
```

```
#include <iostream>
#include "ClassePonto.h"

Ponto::Ponto ()
{
    std::cout << "Objeto Ponto criado!\n";
}

double Ponto::getX (void)
{
    return x;
}

double Ponto::getY (void)
{
    return y;
}

void Ponto::setX (double a)
{
    x = a;
}

void Ponto::setY (double b)
{
    y = b;
}
```

```
// Main
```

```
#include <iostream>
#include "ClassePonto.h"
```

```
using namespace std;
```

```
int main()
{
```

```
    Ponto P1;
    std::cout <<"P1: (" << P1.getX() << ", "
    << P1.getY() << ")\n";
```

```
    return 0;
}
```

```
P1: <0,0>
```

```
// Ponto.h
```

```
class Ponto {
public:
    Ponto(double = 0, double = 0);
    void setX (double);
    void setY (double);
    void setXY (double, double);
    double getX (void);
    double getY (void);
private:
    double x,y;
};
```

```
// Ponto.cpp
```

```
#include <iostream>
#include "ClassePonto.h"
```

```
Ponto::Ponto (double a, double b)
{
    setX(a);
    setY(b);
}
double Ponto::getX (void){
    return x;
}
double Ponto::getY (void){
    return y;
}
void Ponto::setX (double a){
    x = a;
}
void Ponto::setY (double b){
    y = b;
}
```

Objetos `const` e Funções-membro `const`

Objetos const

- Alguns objetos precisam ser modificáveis e alguns não. O programador pode utilizar a palavra chave **const** para especificar que um objeto não pode ser modificado e que qualquer tentativa de modificação gerará um erro de compilação.

Exemplo: `const Ponto Origem;`

Objetos const

- A “constância” de um objeto só é imposta após a inicialização do objeto pelo construtor e é encerrada quando o destrutor é chamado;

Funções-membro const

- Os compiladores C++ não permitem a chamada de funções membro para objetos const, a menos que a própria função membro também seja declarada como const. Isso é válido até para funções como get que são apenas de leitura.
- O compilador não permite que funções membro declaradas como const modifique o objeto.

Funções-membro constantes

- Declarar funções membro como `const` reforça o controle efetuado pelo compilador e permite uma programação mais segura, sem nenhum custo adicional de execução.
- Para definir uma função membro de dados com `const` deve-se colocar a palavra `const` após a lista de parâmetro da função. Veja o exemplo abaixo:
Exemplo: `void imprimePonto () const;`

Exemplo

```
class Ponto {  
    public:  
        Ponto(double = 0, double = 0);  
        void setX (double);  
        void setY (double);  
        void setXY (double, double);  
        double getX (void) const;  
        double getY (void) const;  
        double distanciaDaOrigem (void) const;  
    private:  
        double x,y;  
  
};
```

Exercício

- Declare uma classe Data com os membro de dados dia, mes e ano private;
- Declare funções set e get publicas para que o usuário da classe possa acessar dia, mes, ano;
- Declare um construtor que obrigatoriamente recebe argumentos para inicializar dia, mes e ano.
- Declare uma função public responsável por imprimir a data.
- Declare as funções que não modificam os membros de dados como const;
- Declare um objeto data const e chame a função para imprimir a data.