Composição e Herança

Composição: Objetos como membros de Classes

- Composição
  - Classe pode ter objetos de outras classes como membros
- Construção de objetos
  - Membros são criados na ordem em que são declarados


```
// DEFINICAO DA CLASSE DATA

#ifndef DATA_H
#define DATA_H

class Data {
public:
   Data( int = 1, int = 1, int = 1900 ); // construtor padrão
   void imprime() const;  // imprime data
   ~Data(); //destrutor
private:
   int mes;
   int dia;
   int ano;

// verifica dia
   int verifica_dia( int ) const;
};
#endif
```



```
// EXEMPLO DE COMPOSIÇAO
// DECLARACAO DA CLASSE EMPREGADO.
#ifndef EMPREGADO_H
#define EMPREGADO_H
// incluir Data
#include "data.h"
class Empregado
{
   public:
      Empregado( const char *, const char *, const Data &, const Data & );
      void imprimir() const;
      ~Empregado();
   private:
      char prim_nome[ 25 ];
      char ult_nome[ 25 ];
      const Data nascimento;  // composição
      const Data contratacao;  // composição
   };
#endif
```

Usando composição;


```
// DEFINIÇÃO

#include <iostream>

using std::cout;
using std::endl;

#include <cstring>      // strcpy E strlen

#include "empregado.h"  // Empregado
#include "data.h"       // Data
```



```
// construtor utiliza lista para inicializar valores dos objetos nascimento e
// contratacao
// [Note que ocorre a chamada do construtor de cópia padrão
// que em C++ é fornecido implicitamente pelo compilador.]
Empregado::Empregado( const char *first, const char *last,
   const Data &dateOfBirth, const Data &dateOfHire )
   : nascimento( dateOfBirth ),   // inicializa nascimento
     contratacao( dateOfHire )    // inicializa contratacao
{
   // copia first para prim_nome
   int tam = strlen( first );
   tam = ( tam < 25 ? tam : 24 );
   strncpy( prim_nome, first, tam );
   prim_nome[ tam ] = '\0';
```

```
// PROGRAMA PRINCIPAL
#include <iostream>
using std::cout;
using std::endl;

#include "empregado.h"

int main()
{
   Data birth( 7, 24, 1949 );
   Data hire( 3, 12, 1988 );
   Empregado gerente( "Ricardo", "Silva", birth, hire );

   return 0;
} // fim
```

Cria objeto `Data` para passar data para construtor `Empregado`.

Herança

Introdução

Classes Base e Derivadas

Membros `protected`

Relação entre Classes Base e Derivadas

Construtores e Destrutores em classes Derivadas

Herança Múltipla

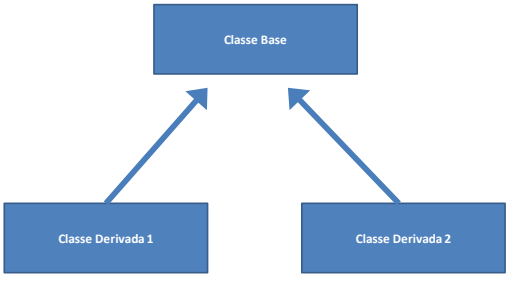
Introdução

- Herança
  - Reutilização de Software
  - Possibilita criar novas classes à partir de outras já existentes
    - Absorve comportamentos e dados
    - Aprimora com novas capacidades

Classes bases x derivadas

- Classes derivadas herdam características da classe base
  - Classes Derivadas
    - São mais especializadas
    - Comportamentos são herdados da classe base
      - » Podem ser modificados
    - Possibilidade de inserir novos comportamentos

Herança



Introdução

- "é-um" x "tem-um"
  - "é-um"
    - Herança
    - Objetos da classe derivada tratados como objetos da classe base
    - Exemplo: carro *é um* veículo
      - Propriedades e comportamentos de veículos são também propriedades e comportamentos de carros
  - "Tem-um"
    - Composição
    - Objeto contém um ou mais objetos de outras classes como membro
    - Exemplo: Carro *tem um* motor

Introdução

- Hierarquia de Classes
  - Classe base direta
    - Classe derivada herda explicitamente (um nível hierárquico acima)
  - Classe base indireta
    - Classe derivada herda de dois níveis hierárquicos acima ou mais

Introdução

- Herança Simples
  - Herda de uma única classe base
- Herança Múltipla
  - Herda de múltiplas classes base
    - Classes base podem não ser inter-relacionadas

Classes Base e Derivadas

- Objeto de uma classe "é-um" objeto de outra
  - Exemplo: Retângulo é um quadrilátero.
    - Classe **`Retângulo`** herda da classe **`Quadrilátero`**
    - **`Quadrilátero`**: classe base
    - **`Retângulo`**: classe derivada
- Classe Base normalmente representa um conjunto de objetos maior que as classes derivadas
  - Exemplo:
    - Classe base: **`Veículo`**
      - » Carro, caminhão, lanchas, bicicletas, …
    - Classe derivada: **`Carro`**
      - » Subconjunto mais específico da classe veículo

Classes Base e Derivadas

- Exemplos de herança

Classe Base	Classes Derivada
`Estudante`	`Estudante_graduação` `Estudante_pós_graduaçao`

`Figura`	`Circulo`
	`Triângulo`
	`Retângulo`

`Empréstimo`	`Empréstimo_para_carro`
	`Empréstimo_para_casa_própria`
	`Empréstimo_pessoal`

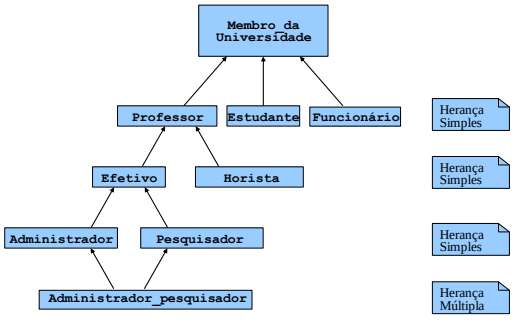
`Empregado`	`Empregado_chão_de_fábrica`
	`Gerente`
	`Diretor`

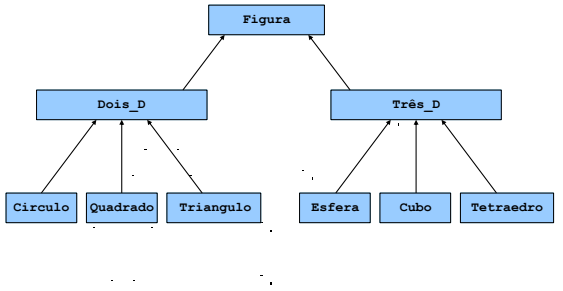
`Conta`	`Conta_corrente` `Conta_poupança`

Classes Base e Derivadas

- Hierarquia de Herança
  - Estrutura de árvore
  - Cada classe se torna
    - Classe Base
      - Compartilha dados e comportamentos c/ outras classes

    OU
    - Classe Derivada
      - Herda dados e comportamentos de outras classes





19

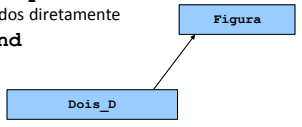
Classes Base e Derivadas

- Três tipos de herança
  - **`public`**
    - Todo objeto da classe derivada é também objeto da classe base
      - Objetos da Classe base não são objetos da classe derivada
      - Exemplo: Todos os carros são veiculos, mas nem todo o veículo é carro (avião, bicicleta...)
    - Pode acessar membros não **`private`** da classe base
  - **`private`**
  - **`protected`**

20

Classes Base e Derivadas

- **`Herança public`**
  - Especificação:
    `Class dois_d : public figura`
    - Classe dois_d herda da classe figura
  - Membros privados da Classe Base
    - Não podem ser acessados diretamente
    - Podem ser acessados através das funções herdadas
  - Membros **`public`** e **`protected`**
    - Podem ser acessados diretamente
  - **`Funções friend`**
    - Não herdadadas



21

Membros `protected`

- **`Acesso protegido`**
  - Nível intermediário de proteção (entre **`public`** e **`private`**)
  - **`Membros protected`** são acessíveis para:
    - Membros da classe base
    - **`Friend`**s da classe base
    - Membros da classe derivada
    - **`Friend`**s da classe derivada

22

Relação entre Classe Base e Derivada

Exemplo: Ponto/circulo

- Ponto
  - Coordenadas <x, y>
- Circulo
  - Coordenadas <x, y>
  - Raio

23

24

SEM herança!

25

```
#ifndef PONTO_H
#define PONTO_H

class Ponto {

public:
    Ponto( int = 0, int = 0 ); // construtor padrão

   void setX( int );
   int getX() const;

   void setY( int );
   int getY() const;

   void print() const;

private:
   int x;
   int y;

};

#endif
```

26

```
#include <iostream>
using std::cout;

// Construtor padrão
Ponto::Ponto( int Valor_x, int Valor_y )
{
   x = Valor_x;
   y = Valor_y;
}

// set x
void Ponto::setX( int Valor_x )
{
   x = Valor_x;
}
```

27

```
// função Get x
int Ponto::getX() const
{
   return x;
}

// set y
void Ponto::setY( int Valor_y )
{
   y = Valor_y;
}

// Get y
int Ponto::getY() const
{
   return y;
}

// impressão
void Ponto::print() const
{
   cout << '[' << x << ", " << y << ']';
}
```

28

```
#include <iostream>
using std::cout;
using std::endl;

#include "ponto.h"
int main()
{
   Ponto ponto( 72, 115 );

   // imprime coordenadas x y
   cout << "Coordenada X é " << ponto.getX()
        << "\nCoordenada Y é " << ponto.getY();

   ponto.setX( 10 ); // set x
   ponto.setY( 10 ); // set y

   // imprime novas coordenadas
   cout << "\n\nA nova posição é ";
   ponto.print();
   cout << endl;
   return 0;
}
```

29

```
Coordenada X é 72
Coordenada Y é 115

A nova posição é [10, 10]
```

30

```
#ifndef CIRCULO_H
#define CIRCULO_H

class Circulo {

public:

   // construtor padrão
   Circulo( int = 0, int = 0, double = 0.0 );

   void setX( int );
   int getX() const;

   void setY( int );
   int getY() const;

   void setRaio( double );
   double getRaio() const;

   double getDiametro() const;
   double getCircunferencia() const;
   double getArea() const;
```

31

```
   void print() const;

private:
   int x;
   int y;
   double raio;

};

#endif
```

32

```
#include <iostream>

using std::cout;

 // construtor padrão
Circulo::Circulo( int V_x, int V_y, double V_raio )
{
   x = V_x;
   y = V_y;
   setRaio( V_raio );
}

// set x
void Circulo::setX( int Valor_x )
{
   x = Valor_x;
}
```

33

```
// Get X
int Circulo::getX() const
{
   return x;
}

// set y
void Circulo::setY( int Valor_y )
{
   y = Valor_y;
}

// Get Y
int Circulo::getY() const
{
   return y;
}
```

34

```
// set raio
void Circulo::setRaio( double Valor_raio )
{
   raio = ( Valor_raio < 0.0 ? 0.0 : Valor_raio );
}

// Get raio
double Circulo::getRaio() const
{
   return raio;
}

// calcula diametro
double Circulo::getDiametro() const
{
   return 2 * raio;
}
```

35

```
// calcula circunferencia
double Circulo::getCircunferencia() const
{
   return 3.14159 * getDiametro();
}

// calcula area
double Circulo::getArea() const
{
   return 3.14159 * raio * raio;
}

// imprime circulo
void Circulo::print() const
{
   cout << "Centro = [" << x << ", " << y << ']'
        << "; Raio = " << raio;
}
```

36

```
 #include <iostream>
 using std::cout;
 using std::endl;
 using std::fixed;

 #include <iomanip>

 using std::setprecision;

 #include "circulo.h"  // definição de circulo

int main()
{
   Circulo c( 37, 43, 2.5 );

   cout << "Coordenada X " << c.getX()
        << "\nCoordenada Y " << c.getY()
        << "\nRaio " << c.getRaio();
```

37

```
    c.setX( 2 );
    c.setY( 2 );
    c.setRaio( 4.25 );

    // imprimir novo ponto
    cout << "\n\nDados do novo circulo \n";
    c.print();

    // formatar saida
    cout << fixed << setprecision( 2 );

    // imprimir diametro do circulo
    cout << "\nDiametro : " << c.getDiametro();

    // imprimir circunferencia do circulo
    cout << "\nCircunferencia :  " << c.getCircunferencia();

    // Area
    cout << "\nArea : " << c.getArea();
    cout << endl;
    return 0;
}
```

38

Usando herança!

39

```
#ifndef NOVO_CIRCULO_H
#define NOVO_CIRCULO_H

#include "ponto.h"

class Novo_circulo : public Ponto {

public:
   // construtor padrao
   Novo_circulo( int = 0, int = 0, double = 0.0 );

   void setRaio( double );
   double getRaio() const;

   double getDiametro() const;
   double getCircuferencia() const;
   double getArea() const;

   void print() const;

private:
   double raio;
};

#endif
```

Classe Novo_circulo herda da classe **Ponto**.

: indica herança.

Palavra-chave **public** indica tipo de herança.

Raio é membro privado

40

```
#include <iostream>

using std::cout;

Novo_circulo::Novo_circulo( int Vx, int Vy, double Vraio )
{
   x = Vx;
   y = Vy;
   setRaio( Vraio );
}
```

Tentativa de acessar membros privados da classe base resulta em erro de sintaxe.

41

```
// set raio
void Novo_Circulo::setRaio( double V_raio )
{
   raio = ( V_raio < 0.0 ? 0.0 : V_raio );
}

double Novo_Circulo::getRaio() const
{
   return raio;
}

double Novo_Circulo::getDiametro() const
{
   return 2 * raio;
}
```

42

```
// calcula circunferencia
double Novo_circulo::getCircuferencia() const
{
   return 3.14159 * getDiametro();
}

// calcula area
double Novo_circulo::getArea() const
{
   return 3.14159 * raio * raio;
}

// imprime objeto
void Novo_circulo::print() const
{
   cout << "Centro = [" << x << ", " << y << ']'
        << "; Raio = " << raio;
}
```

Novamente erro de sintaxe por tentar acessar membro privado da classe base.

43

Exemplo: Usando *protected*

44

```
#ifndef POINT2_H
#define POINT2_H
class Novo_ponto {

public:
   Novo_ponto( int = 0, int = 0 ); // construtor

   void setX( int );
   int getX() const;

   void setY( int );
   int getY() const;

   void print() const;

protected:
   int x;
   int y;

};

#endif
```

Dados protected são acessíveis às classes derivadas.

45

```
#include <iostream>
using std::cout;
Novo_circulo::Novo_circulo( int Vx, int Vy, double Vr )
{
   x = Vx;
   y = Vy;
   setRaio( Vr );
} // end Circle3 constructor

void Novo_circulo::setRaio( double Vr )
{
   raio = ( Vr < 0.0 ? 0.0 : Vr );
}
```

Pode modificar os dados declarados como protected na classe base

Construtor chama primeiramente (de forma implícita) o construtor da classe base

Relação entre Class Base e Derivada

- Usando dados **`protected`**
  - Vantagens
    - Classes derivadas podem modificar valores diretamente
  - Desvantagens
    - Sem validação
      - Classes Derivadas podem atribuir valores ilegais
    - Implementação dependente da classe base
      - Mudanças no código da classe base podem resultar em modificações nas classes derivadas

46

47

```
#ifndef POINT3_H
#define POINT3_H

class Ponto {

public:
   Ponto( int = 0, int = 0 );

   void setX( int );
   int getX() const;

   void setY( int );
   int getY() const;

   void print() const;

private:
   int x;
   int y;

};
#endif
```

Melhor prática de engenharia de software: prefira **private**.

48

```
#include <iostream>

using std::cout;

#include "ponto.h"

// default constructor
Ponto::Ponto( int Vx, int Vy ) : x( Vx ), y( Vx )
{
   // vazio
}

void Ponto::setX( int Vx )
{
   x = Vx;
}
```

Inicializadores especificam valores de **x** e **y**.

49

```
int Ponto::getX() const
{
    return x;
 }

void Ponto::setY( int Vy )
{
   y = Vy;
}

int Ponto::getY() const
{
   return y;
}

void Ponto::print() const
{
   cout << '[' << getX() << ", " << getY() << ']';
}
```

Utilizando funções de acesso aos membros private.

50

```
#ifndef CIRCULO_H
#define CIRCULO_H

#include "ponto.h"

class Circulo : public Ponto {
public:
  Circulo( int = 0, int = 0, double = 0.0 );

  void setRaio( double );
  double getRaio() const;

  double getDiametro() const;
  double getCircunferencia() const;
  double getArea() const;
  void print() const;

private:
  double raio;

};

#endif
```

Circulo herda de ponto

51

```
#include <iostream>
using std::cout;
#include "circulo.h"

Circulo::Circulo( int Vx, int Vy, double Vr )
   : Ponto( Vx, Vy )  // construtor da classe base
{
   setRaio( Vr );
}

void Circulo::setRaio( double Vr )
{
   raio = ( Vr < 0.0 ? 0.0 : Vr );
}
```

52

```
double Circulo::getRaio() const
{
   return raio;
}

double Circulo::getDiametro()
{
   return 2 * getRaio();
}

double Circulo::getCircunferencia() const
{
   return 3.14159 * getDiametro();
}
```

É melhor acessar raio através da função get do que diretamente.

53

```
double Circulo::getArea() const
{
   return 3.14159 * getRaio() * getRaio();
}

void Circulo::print() const
{
   cout << "Centro = ";
   Ponto::print(); // chamando função de impressão do ponto
   cout << "; Raio = " << getRaio();
}
```

Exemplo
Tres níveis hierárquicos de herança

Ponto

Circulo

Cilindro

54

55

```
#ifndef CILINDRO_H
#define CILINDRO_H
#include "circulo.h"

class Cilindro : public Circulo
{
 public:
 // construtor padrão
 Cilindro( int = 0, int = 0, double = 0.0, double = 0.0 );
  void setAltura( double );
   double getAltura() const;
   double getArea() const;
   double getVolume() const;
   void print() const;

 private:
   double altura;

};
#endif
```

Classe **Cilindro** herda "características" da classe **Circulo**.

Altura permanece **private**.

56

```
#include <iostream>

using std::cout;

#include "cilindro.h"

// construtor padrão
Cilindro::Cilindro( int Vx, int Vy, double Vr,
  double Vh )
  : Circulo( Vx, Vy, Vr )
{
   setAltura( Vh );
}
```

57

```
void Cilindro::setAltura( double Vh )
{
   Altura = ( Vh < 0.0 ? 0.0 : Vh );
}

double Cilindro::getAltura() const
{
   return altura;
}

double Cilindro::getArea() const
{
   return 2 * Circulo::getArea() +
      getCircunferencia() * getAltura();
}
```

Redefinição da função getArea ( ) para retornar a área do cilindro e não a do círculo.

58

```
double Cilindro::getVolume() const
{
   return Circulo::getArea() * getAltura();
}

void Cilindro::print() const
{
   Circulo::print();
   cout << "; Altura = " << getAltura();
}
```

Redefinição da função print

Exercício

- Criar classe Conta, para representar contas bancárias, que deve ter:
  - métodos *debitar e creditar (com validação)*
  - atributo *Saldo (com métodos get e set)*.
- Usando herança, crie os tipos mais específicos de conta:
  - Poupança
    - Atributo: Juros
    - Método que retorna juros auferidos
  - ContaCorrente
    - Atributo: Taxa cobrada por por transações
      - Metodos debitar e creditar devem diminuir saldo

59

Construtores e Destrutores em Classes Derivadas

- Instanciação de objetos de classes derivadas
  - Cadeia de chamadas de construtores
    - Construtor da classe derivada invoca construtor da classe base
      - Implícita ou explicitamente
    - Base da hierarquia
      - Último construtor chamado
      - Primeiro construtor a finalizar a execução
      - Exemplo: **Ponto/Circulo/Cilindro**
        - » **Ponto** é o último construtor chamado
        - » **Ponto** é o primeiro construtor a terminar execução

60

Construtores e Destrutores em Classes Derivadas

- Destruindo objetos de classes derivadas
  - Cadeia de chamadas de destrutores
    - Ordem reversa dos construtores
    - Destrutor da classe derivada chamado primeiramente
    - Destrutor da próxima classe base chamado
      - Continua até atingir a classe base no topo da hierarquia
        - » Depois de finalizada a chamada em cadeia o objeto é removido da memória


```
#ifndef PONTO_H
#define PONTO_H

class Ponto {

public:
   Ponto( int = 0, int = 0 );
  ~Ponto();

   void setX( int );
   int getX() const;
   void setY( int );
   int getY() const;
   void print() const;

private:
   int x;
   int y;

};

#endif
```

Para demonstrar a ordem da chamada dos construtores e destrutores pode-se colocar mensagens.


```
#include <iostream>
using std::cout;
using std::endl;
#include "pointo.h"

Ponto::Ponto( int Vx, int Vy )
  : x( Vx ), y( Vy )
{
  cout << "Construtor de Ponto ";
  print();
  cout << endl;
}

Ponto::~Ponto()
{
  cout << "Destrutor de Ponto ";
  print();
  cout << endl;
}
```



```
void Ponto::setX( int Vx )
{
  x = Vx;
}
int Ponto::getX() const
{
  return x;
}
void Ponto::setY( int Vy )
{
  y = Vy;
}
int Ponto::getY() const
{
 return y;
}
void Ponto::print() const
{
 cout << '[' << getX() << ", " << getY() << ']';
}
```



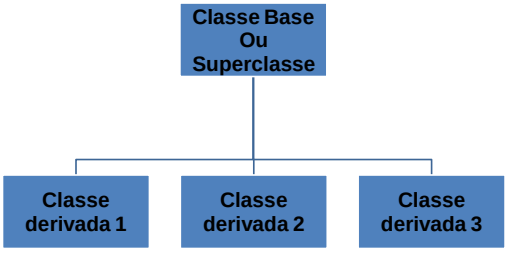
```
#include <iostream>

using std::cout;
using std::endl;

#include "circulo.h"

int main()
{
   { // inicio de novo escopo
        Ponto ponto( 11, 22 );
   } // fim do escopo

   cout << endl;
   Circulo circulo1( 72, 29, 4.5 );
   cout << endl;
   Circulo circulo2( 5, 5, 10 );
   cout << endl;

   return 0;
} // Fim do programa principal
```



```
Construtor de Ponto [11, 22]
Destrutor de Ponto  [11, 22]

Construtor de Ponto [72, 29]
Construtor de Circulo Centro = [72, 29]; Raio = 4.5

Construtor de Ponto [5, 5]
Construtor de Circulo Centro = [5, 5]; Raio = 10

Destrutor de Circulo Centro = [5, 5]; Raio = 10
Destrutor de Ponto  [5, 5]
Destrutor de Circulo Centro = [72, 29]; Raio = 4.5
Destrutor de Ponto  [72, 29]
```

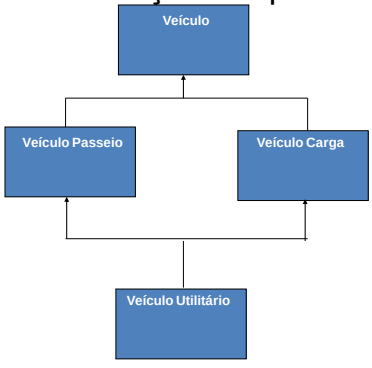
Tipo de acesso do membro da classe base	Tipo de Herança		
	`public`	`protected`	`private`
`Public`	`public` na classe derivada. Pode ser acessada diretamente por qualquer função membro não `static`, funções `friend` e funções não membros.	`protected` na classe derivada. Pode ser acessada diretamente por todas as funções membro não `static` e funções `friend`.	`private` na classe derivada. Pode ser acessada diretamente por todas funções membro não `static` e funções `friend`.
`Protected`	`protected` na classe derivada. Pode ser acessada diretamente por todas as funções membro não `static` e funções `friend`.	`protected` na classe derivada. Pode ser acessada diretamente por todas as funções membro não `static` e funções `friend`.	`private` na classe derivada. Pode ser acessada diretamente por todas as funções membro não `static` e funções `friend`.
`Private`	Escondida na classe derivada. Pode ser acessada por funções não `static` e funções `friend` através de funções membro `public` ou `protected` da classe base. .		

Herança



68

Herança Múltipla



69

70

```
class Veiculo {

 private:
   char *nome;
   int peso, hp;

 public:

   veiculo( char *n, int p, int h );

   void set_hp  ( int );
   int  get_hp() const;
   void set_peso( int );
   int  get_peso() const;
   void print() const;

};
```

71

```
class v_passeio : public veiculo {

 private:
   int  volume_interno;

 public:

   v_passeio( char *n, int p, int hp, int vol_int );

   void set_vi  ( int );
   int  get_vi() const;
   float peso_potencia();
   void print() const;
};
```

72

```
class v_carga : public veiculo {

 private:
   int  carga;

 public:

   v_carga( char *n, int p, int hp, int carga );

   void set_carga  ( int );
   int  get_carga() const;
   float peso_potencia();
   void print() const;
};
```

```
class v_utilitario : public v_passeio, public v_carga {

 private:
   int  qdade_pessoas;

 public:

    v_utilitario( char *n, int p, int hp, int vi, int carga );

   float peso_potencia();
   void print() const;
};
```

73

```
veiculo::veiculo (char *n, int p, int h)
{
  nome = n; peso = p; hp = h;
}
void veiculo::set_peso(int p)
{
  peso = p;
}
void veiculo::get_peso()
{
  return peso;
}
...
v_passeio::v_passeio(char *n, int p, int hp, int vi)
  :veiculo(n,p,jp)
{
  volume_interno = vi;
}
...
```

74

```
v_passeio::peso_potencia ()
{
  return float(get_peso())/float(get_hp());
}
v_carga::peso_potencia()
{
  return float(get_peso()+carga)/float(get_hp());
}
v_utilitario::peso_potencia()
{
  return v_carga::peso_potencia();
}
```

75

Conceitos Finais

- Sobrecarregando x Anulando
  - Sobrecarga → mais de um método com o mesmo nome, mas com assinaturas diferentes.
  - Anulação → método em uma classe derivada com o mesmo nome que método na classe base e mesma assinatura.

76