Aula 8:

Polimorfismo
e Funções Virtuais

ECOP13A - Programação Orientada a Objetos
Prof. André Bernardi
andrebernardi@unifei.edu.br

Universidade Federal de Itajubá

Rev. maio 25

Polimorfismo

O dicionário define a palavra **polimorfismo** como um princípio da biologia no qual uma mesma espécie de organismo pode apresentar **várias formas diferentes**.

> Pode ser utilizado também na programação, e está intimamente relacionado ao processo de generalização, ou herança. Este princípio torna a programação orientada a objeto muito poderosa.

Através dele, objetos de classes diferentes de uma mesma hierarquia de classes podem responder a uma mesma mensagem de maneiras diferentes.

Polimorfismo

Mas como assim?

Polimorfismo permite escrever programas que processam objetos de classes que fazem parte da mesma hierarquia de classes como se fossem todos objetos da *classe base* da hierarquia.

Em um programa que simula a movimentação de vários tipos de animais para um estudo biológico, as classes **Passaro**, **Peixe** e **Sapo** representam três animais sob investigação e cada uma delas herda da mesma classe base **Animal**, que contém uma função **mover** e dados com a posição atual. Cada classe derivada implementa a função mover. Nosso programa mantém um vetor ponteiros para animais. Para simular a movimentação, é necessário chamar a função **mover** de cada um dos animais do vetor uma vez por segundo. Ao chamar a função mover da referência para Animal (classe base), cada objeto irá responder de uma maneira específica, de acordo com a implementação de sua classe derivada. **Mesma mensagem (**mover**), resultados diferentes.**

Polimorfismo

Este conceito, portanto, permite que um programa seja escrito de maneira mais geral, possibilitando que sejam implementados sistemas **facilmente extensíveis**.

> Para incluir, por exemplo, uma classe **Tartaruga** ao programa anterior, bastaria que ela também herdasse de animal e implementasse a função **mover** de sua própria maneira. A partir daí, toda a simulação de movimentação no vetor de animais poderia contar também com tartarugas, que mover-se-iam bem devagar.

As partes da simulação que processam cada Animal genericamente poderiam permanecer iguais.

Interações entre objetos em uma hierarquia de classes

Na última aula aprendemos como realizar atribuições entre objetos de uma mesma hierarquia de classes. São duas regras básicas:

- A atribuição de um objeto de uma **subclasse** para sua **superclasse** é possível, mas "*trunca*" o objeto, mantendo apenas os dados e comportamentos pertencentes à superclasse.
- A atribuição de um objeto da **superclasse** para a subclasse não é possível e gera erro de compilação.

Veja exemplos:

```
Professor pf{"Steve", "Washington", 134515, 2, "MIT"};
Pessoa p2 = pf; // Pode isso? SIM. Mas info específica de Professor se perde.
Professor pf2 = p2; // Pode isso? NÃO!
Professor pf3 = (Professor)p2; // E isso? Também NÃO!
```

Interações entre objetos em uma hierarquia de classes

No entanto, precisamos também saber o que ocorre quando tentamos atribuir ponteiros entre classes de uma mesma hierarquia. O que acontece quando:

- Criamos um ponteiro da classe **base** e o apontamos para a própria classe **base**? Base → Base
- Criamos um ponteiro da classe **derivada** e apontamos também para um objeto da classe **derivada**? Derivada → Derivada
- Criamos um ponteiro da classe **base** e o apontamos para um objeto de uma classe **derivada**? Base → Derivada
- No último caso, teríamos acesso as funções membro da classe derivada?
- Criamos um ponteiro da classe **derivada** e o apontamos para um objeto de uma classe **base**? Derivada → Base

Vejamos as repostas:

Os casos básicos funcionam normalmente, quando temos ponteiros do tipo exato do objeto para o qual aponta. Veja:

```
int main()
{
    Pessoa p{"Joao", "Itajuba"};
    Professor pf{"Pedro", "SJC", 132515, 2, "IESTI"};

    Pessoa *ptrJoao = &p; // ponteiro para Pessoa aponta para Pessoa.
    Professor *ptrPedro = &pf; // Ponteiro para Professor aponta para Professor

    cout << "Pessoa: \n";
    ptrJoao->imprime_perfil(); // utilizamos o operador -> ao invés de .
    cout << "\nProfessor: \n";
    ptrPedro->imprime_perfil(); // cada um chama sua própria função imprime_perfil
}
```

```
Pessoa:
Nome: Joao
Endereco: Itajuba
Professor:
Nome: Pedro
Endereco: SJC
siape: 132515
categoria: 2
instituto: IESTI
```

ptrJoao chama a função **Pessoa**::imprime_perfil(), enquanto **ptrPedro** chama **Professor**::imprime_perfil().

Podemos criar um ponteiro para a classe **base** e apontá-lo para um objeto de uma **derivada** sua. Funciona normalmente ("**é um**"), mas acessa apenas dados e funções públicas da classe base. Veja:

```
int main()
{
    Pessoa p{"Joao", "Itajuba"};
    Professor pf{"Pedro", "SJC", 132515, 2, "IESTI"};

    Pessoa *ptrJoao = &p; // ponteiro para Pessoa aponta para Pessoa.
    Pessoa *ptrPedro = &pf; // Ponteiro para Pessoa aponta para Professor

    cout << "Pessoa: \n";
    ptrJoao->imprime_perfil(); // utilizamos o operador -> ao invés de .
    cout << "\nProfessor: \n";
    ptrPedro->imprime_perfil(); // os dois chamam Pessoa::imprime_perfil
}
```

ptrJoao chama a função **Pessoa**::imprime_perfil(),
e **ptrPedro** também chama **Pessoa**::imprime_perfil().

```
Pessoa:
Nome: Joao
Endereco: Itajuba
Professor:
Nome: Pedro
Endereco: SJC
```

Guarde isso!

Finalmente, **não é possível** fazer com que um ponteiro para uma classe derivada aponte para um objeto da classe base. O compilador geraria erro, dizendo que não é possível converter os valores.

```
int main()
{
    Pessoa p{"Joao", "Itajuba"};
    Professor pf{"Pedro", "SJC", 132515, 2, "IESTI"};

    Professor *ptrJoao = &p; // ponteiro para Professor aponta para Pessoa.
    Professor *ptrPedro = &pf; // Ponteiro para Professor aponta para Professor

    cout << "Pessoa: \n";
    ptrJoao->imprime_perfil();
    cout << "\nProfessor: \n";
    ptrPedro->imprime_perfil();
}
```

```
principal.cpp: In function 'int main()':
principal.cpp:10:24: error: invalid conversion from 'Pessoa*' to 'Professor*' [-fpermissive]
  Professor *ptrJoao = &p; // ponteiro para Professor aponta para Pessoa.
                       ^
```

Podemos “forçar” uma conversão de tipos neste último caso, mas ela só irá funcionar corretamente caso o ponteiro para a classe base esteja apontando para um objeto da classe para a qual está sendo convertido. O processo é conhecido como ***downcasting***.

```
int main()
{
    Pessoa p{"Joao", "Itajuba"};
    Professor pf{"Pedro", "SJC", 132515, 2, "IESTI"};

    Pessoa *ptrJoao = &p; // ponteiro para Professor aponta para Pessoa.
    Pessoa *ptrPedro = &pf; // Ponteiro para Professor aponta para Professor
    Professor* ptrProf = (Professor*) ptrPedro; // conversão explícita (downcasting)

    cout << "Pessoa: \n";
    ptrJoao->imprime_perfil(); // Pessoa::imprime_perfil()
    cout << "\nProfessor: \n";
    ptrProf->imprime_perfil(); // Professor::imprime_perfil()
}
```

```
Pessoa:
Nome: Joao
Endereco: Itajuba
Professor:
Nome: Pedro
Endereco: SJC
siape: 132515
categoria: 2
instituto: IESTI
```

Caso **ptrPedro** não contivesse um endereço para um objeto da classe Professor, seu programa funcionaria de maneira imprevisível.

Evite este tipo de risco!

Funções Virtuais

Sabemos, agora, que um ponteiro genérico de uma classe **base** pode receber o endereço para objetos de qualquer uma de suas classes **derivadas**. Se houver uma classe Pessoa, um ponteiro para Pessoa poderia conter endereços para Alunos, Professores, etc.

No entanto, ao acessar os membros através do **ponteiro do tipo base**, somente conseguimos acessar funções membro da classe base, mesmo que elas tenham sido redefinidas na classe derivada.

Importante: Não é o tipo do objeto apontado que determina a versão da função a ser chamada, mas sim o tipo do ponteiro que aponta para ele.

Funções Virtuais

E se quiséssemos que fosse ao contrário, ou seja, que a definição da função a ser chamada seja realizada com base no objeto apontado e não no tipo do ponteiro? **Isso é polimorfismo**!

> Lembre-se: Objetos diferentes respondendo a mesma mensagem (chamada de função) de maneiras diferentes.
>
> Isso poderá ser alcançado a partir da criação de **funções virtuais**.

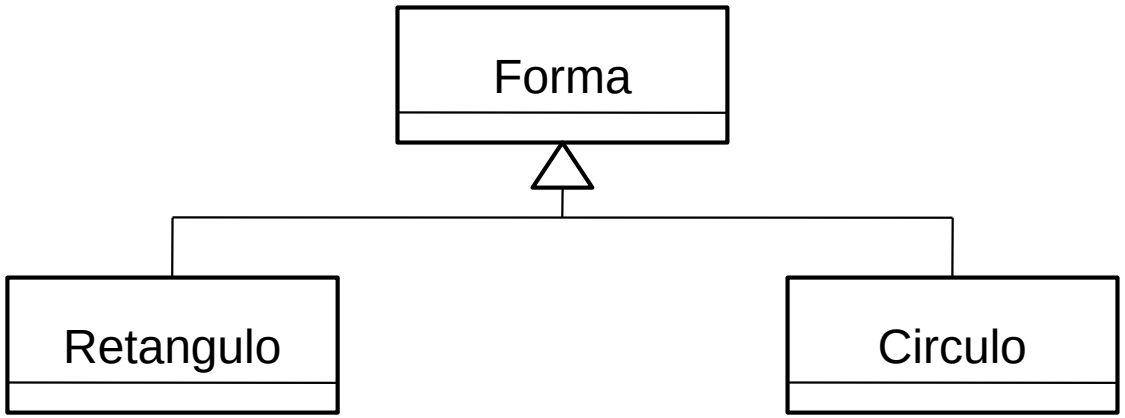
Ao declarar uma função como **virtual** na classe base, podemos sobrepô-la (*override*) nas classes derivadas. Do ponto de vista da implementação, sobrepor uma função não é diferente de redefini-la: sua assinatura é idêntica à função base. **Isso vai garantir o polimorfismo**.

Funções Virtuais

Para declarar uma função virtual, devemos incluir a palavra reservada *virtual* antes de seu tipo de retorno. Veja um exemplo:

```
virtual double area() const;
```

Vamos ver exemplos de código que tornam mais clara a sua utilização. Faremos duas funções virtuais: **área** e **imprime_dados**.



```
class Forma {
    protected:
        double x, y;
    public:
        Forma(double=0, double=0);
        ~Forma() {}

        // Funções que podem ser "sobrepostas" em classes derivadas
        virtual void imprime_dados();
        virtual double area() { return 0; } // inline
};
```

```
// Construtor
Forma::Forma(double xx, double yy) : x{xx}, y{yy} {}

// Implementação da função de impressão (que é virtual)
void Forma::imprime_dados()
{
    cout << "Origem: (" << x << "," << y << ")" << "\n";
}
```

Para tornar qualquer função virtual (preparada para o polimorfismo), basta incluir a palavra **virtual** antes de seu tipo de retorno.

```
class Retangulo : Forma {
    private:
        double l1, l2;
    public:
        Retangulo(double, double, double, double);
        ~Retangulo() {};

        // sobrepondo funções virtuais da classe base
        void imprime_dados();
        double area();
};
```

Mesmas assinaturas das funções virtuais da classe base.

```
Retangulo::Retangulo(double xx, double yy, double v, double h)
    : Forma{xx,yy}, l1{v}, l2{h} {}

// realiza nova implementação, específica para retangulos
double Retangulo::area()
{
    return l1*l2;
}

// chama função base, mas complementa.
void Retangulo::imprime_dados()
{
    Forma::imprime_dados();
    cout << "Medidas dos lados: " << l1 << " e " << l2;
    cout << "\nArea: " << area() << "\n";
}
```

Implementações personalizadas

```
class Circulo : Forma {
    private:
        double raio;
    public:
        Circulo(double, double, double);
        ~Circulo() {}

        // continuam como virtuais, para subclasses de
        // Circulo, como, digamos, esfera ou cilindro.
        virtual double area();
        virtual void imprime_dados();
};
```

```
Circulo::Circulo(double xx, double yy, double r)
    : Forma{xx,yy}, raio{r} {}

double Circulo::area()
{
    return 3.141592*raio*raio;
}

void Circulo::imprime_dados()
{
    Forma::imprime_dados();
    cout << "Raio: " << raio << "\n";
    cout << "Area: " << area() << "\n";
}
```

As funções que sobrepõe outras também podem ser **virtuais**, para o caso de que a classe derivada sirva de base para outras.

```
int main()
{
    Forma *ponto = new Forma{100,100};
    cout << "Ponto ponto: \n";
    ponto->imprime_dados();
    cout << "\n";

    Retangulo *ret = new Retangulo{10,10,20,30};
    cout << "Retangulo ret: \n";
    ret->imprime_dados();
    cout << "\n";

    Circulo *circ = new Circulo{10,10,10};
    cout << "Circulo circ: \n";
    circ->imprime_dados();
    cout << "\n";
}
```

```
Ponto ponto:
Origem: (100,100)

Retangulo ret:
Origem: (10,10)
Medidas dos lados: 20 e 30
Area: 600

Circulo circ:
Origem: (10,10)
Raio: 10
Area: 314.159
```

Veja que temos ponteiros específicos apontando para objetos de seus próprios tipos. Cada classe irá acessar seus próprios métodos de impressão e cálculo de área. A função main não tem nada de especial.

Funções Virtuais e Polimorfismo

Agora, vamos fazer algumas manipulações com ponteiros e verificar os resultados. Veja uma nova função main:

```
int main()
{
    // ponteiro para classe base apontando para objeto de classe base
    Forma *ponto = new Forma{100,100};
    cout << "Ponto ponto: \n";
    ponto->imprime_dados();
    cout << "\n";

    // ponteiro para classe base apontando para objeto de derivada
    Forma *ret = new Retangulo{10,10,20,30};
    cout << "Retangulo ret: \n";
    ret->imprime_dados(); // o que imprime?
    cout << "\n";

    // ponteiro para classe base apontando para objeto de derivada
    Forma *circ = new Circulo{10,10,10};
    cout << "Circulo circ: \n";
    circ->imprime_dados(); // o que imprime?
    cout << "\n";
}
```

Ponteiro da base

Objetos da Derivada

Com ponteiros do tipo da **classe base** apontando para objetos de classes **derivadas**, o que é impresso em **imprime_dados** em cada um dos casos?

```cpp
int main()
{
    // ponteiro para classe base apontando para objeto de classe base
    Forma *ponto = new Forma{100,100};
    cout << "Ponto ponto: \n";
    ponto->imprime_dados();
    cout << "\n";

    // ponteiro para classe base apontando para objeto de derivada
    Forma *ret = new Retangulo{10,10,20,30};
    cout << "Retangulo ret: \n";
    ret->imprime_dados(); // o que imprime?
    cout << "\n";

    // ponteiro para classe base apontando para objeto de derivada
    Forma *circ = new Circulo{10,10,10};
    cout << "Circulo circ: \n";
    circ->imprime_dados(); // o que imprime?
    cout << "\n";
}
```

```
Ponto ponto:
Origem: (100,100)

Retangulo ret:
Origem: (10,10)
Medidas dos lados: 20 e 30
Area: 600

Circulo circ:
Origem: (10,10)
Raio: 10
```

Exatamente!

Cada uma imprime os dados de sua própria maneira! Quando as funções são virtuais, a função a ser chamada é escolhida com base no objeto para o qual o ponteiro aponta e não no tipo de ponteiro. Isso é chamado de **vinculação dinâmica** ou **resolução dinâmica de métodos**, e possibilita o **polimorfismo**.

Objetos diferentes respondendo de maneira diferentes à mesma mensagem = **Polimorfismo**.

Funções Virtuais e Polimorfismo

Façamos algo ainda mais interessante:

```
int main()
{
    // ponteiro para classe base apontando para objeto de classe base
    Forma *ponto = new Forma{100,100};
    Forma *ret1 = new Retangulo{10,10,20,30};
    Forma *ret2 = new Retangulo{0,0,2,5};
    Forma *circ1 = new Circulo{10,10,10};
    Forma *circ2 = new Circulo{0,1,12};

    // Um vetor de ponteiros para formas pode receber ponteiros
    // para qualquer um dos derivados de forma, inclusive outros que
    // ainda estão para ser criados, como Triangulo.
    Forma* vetor_formas[5];
    vetor_formas[0] = circ1;
    vetor_formas[1] = ret1;
    vetor_formas[2] = circ2;
    vetor_formas[3] = ret2;
    vetor_formas[4] = ponto;

    // for imprime cada um dos dados à sua maneira
    for(int i = 0; i < 5; i++)
    {
        cout << "Forma " << i+1 << ": \n";
        vetor_formas[i]->imprime_dados(); // vinculação dinâmica
        cout << endl;
    }
}
```

Podemos criar conjuntos de dados genéricos, do tipo da classe mais geral da hierarquia, e, através de funções virtuais, fazer com que cada elemento responda às chamadas de funções de maneira específica, de acordo com seu próprio tipo.

```cpp
int main()
{
    // ponteiro para classe base apontando para objeto de classe base
    Forma *ponto = new Forma{100,100};
    Forma *ret1 = new Retangulo{10,10,20,30};
    Forma *ret2 = new Retangulo{0,0,2,5};
    Forma *circ1 = new Circulo{10,10,10};
    Forma *circ2 = new Circulo{0,1,12};

    // Um vetor de ponteiros para formas pode receber ponteiros
    // para qualquer um dos derivados de forma, inclusive outros que
    // ainda estão para ser criados, como Triangulo.
    Forma* vetor_formas[5];
    vetor_formas[0] = circ1;
    vetor_formas[1] = ret1;
    vetor_formas[2] = circ2;
    vetor_formas[3] = ret2;
    vetor_formas[4] = ponto;

    // for imprime cada um dos dados à sua maneira
    for(int i = 0; i < 5; i++)
    {
        cout << "Forma " << i+1 << ": \n";
        vetor_formas[i]->imprime_dados(); // vinculação dinâmica
        cout << endl;
    }
}
```


```
Forma 1:
Origem: (10,10)
Raio: 10
Area: 314.159

Forma 2:
Origem: (10,10)
Medidas dos lados: 20 e 30
Area: 600

Forma 3:
Origem: (0,1)
Raio: 12
Area: 452.389

Forma 4:
Origem: (0,0)
Medidas dos lados: 2 e 5
Area: 10

Forma 5:
Origem: (100,100)
```

Classes Abstratas e Funções Virtuais Puras

Quando pensamos em uma classe como um tipo, supomos que os programas criarão objetos desse tipo. Porém, existem casos em que é útil definir classes das quais você nunca pretende instanciar objetos. Essas classes são chamadas de **classes abstratas**.

> Essas classes serão normalmente utilizadas como base de hierarquias de classes. Elas não podem ser utilizadas para instanciar objetos pois, como veremos, elas são classes incompletas: as classes derivadas é que irão implementar as “partes que faltam”.

Classes que podem ser utilizadas para instanciar objetos (como as que vimos até hoje) são chamadas de **classes concretas**.

Classes Abstratas e Funções Virtuais Puras

Forma é uma classe que bem poderia ser abstrata. O que é uma forma? Qual o seu formato? Dificilmente teríamos utilização prática para objetos desta classe. As classes concretas **Circulo** e **Retangulo** é que tornariam razoável a instanciação de objetos.

Uma hierarquia de herança não precisa conter nenhuma classe abstrata, mas isso é bem comum.

Como definir uma classe abstrata?

Declarando uma ou mais de suas funções virtuais como “puras”, como em:

```
virtual double area() const = 0;          // não implementada na base
```

Classes Abstratas e Funções Virtuais Puras

Uma função virtual pura é especificada colocando-se "**=0**" e **não apresenta implementação**. Uma classe concreta derivada dela precisa obrigatoriamente sobrepor todas as ***funções virtuais puras*** da classe base. A diferença é que em funções virtuais comuns, é apenas dada a opção ao programador de sobrepor a função na classe derivada: pode-se fazer ou não.

Funções virtuais puras são usadas quando não há sentido para sua implementação na classe base, mas você deseja que todas as classes derivadas a implementem. Assim, você cria um padrão de tipos, e comportamentos obrigatórios para classes derivadas. A função **area()** para Forma é um bom exemplo.

```
class Forma {
    protected:
        double x, y;
    public:
        Forma(double=0, double=0);
        ~Forma() {}

        // Funções virtuais
        virtual void imprime_dados(); // virtual comum (pode ser sobreposta)
        virtual double area() = 0; // virtual pura (DEVE ser sobreposta)
};
```

O simples fato de tornar a função virtual pura, transforma a classe Forma em uma classe abstrata. A tentativa de se criar um objeto da classe Forma gera um erro de compilação semelhante a este:

```
principal.cpp: In function 'int main()':
principal.cpp:11:34: error: invalid new-expression of abstract class type 'Forma
'
  Forma *ponto = new Forma{100,100};
                                  ^
In file included from principal.cpp:2:0:
forma.h:7:7: note:   because the following virtual functions are pure within 'Fo
rma':
 class Forma {
       ^
forma.h:16:18: note:    virtual double Forma::area()
   virtual double area() = 0; // virtual pura
                  ^
```

Classes Abstratas e Funções Virtuais Puras

Uma classe abstrata tem **pelo menos uma** função virtual pura mas pode, também, ter membros de dados e funções membro concretas (incluindo construtores e destrutores) que estejam sujeitos a regras normais de herança por classes derivadas.

A classe abstrata define uma interface pública comum a todas as classes de sua hierarquia.

Imagine que queiramos que qualquer forma geométrica saiba, obrigatoriamente, calcular sua própria **área**, **perímetro** e saiba **imprimir seus dados**:

```
class Forma {
    protected:
        double x, y;
    public:
        Forma(double xx=0, double yy=0) : x{xx}, y{yy} {}
        ~Forma() {}

        // Funções virtuais puras
        virtual void imprime_dados() const = 0;
        virtual double area() const = 0;
        virtual double perimetro() const = 0;
};
```

Veja um exemplo de classe derivada concreta de Forma: Circulo.

```
class Circulo : public Forma {
    private:
        double raio;
    public:
        Circulo(double, double, double);
        ~Circulo() {}

        // implementam funções virtuais puras de Forma
        // Precisam ter a mesma assinatura (inclusive const)
        double area() const;
        void imprime_dados() const;
        double perimetro() const;
};
```

Para ser concreta, precisa implementar **todas** as virtuais puras de sua classe base

```
Circulo::Circulo(double xx, double yy, double r)
    : Forma{xx,yy}, raio{r} {}

double Circulo::area() const
{
    return 3.141592*raio*raio;
}

void Circulo::imprime_dados() const
{
    cout << "Origem: (" << x << "," << y << ")" << "\n";
    cout << "Raio: " << raio << "\n";
    cout << "Area: " << area() << "\n";
}

double Circulo::perimetro() const
{
    return 2*3.141592*raio;
}
```

Impressionantemente, é possível chamar o construtor da classe Forma, afinal de contas, não estamos criando um novo objeto do tipo Forma, apenas estamos pedindo para que inicialize seus membros de dados.

A implementação da classe derivada **Retangulo** é semelhante, mas implementa as funções virtuais puras a sua própria maneira.

Destrutores Virtuais

Pode haver problemas quando se usa polimorfismo para processar dinamicamente objetos alocados de uma hierarquia de classes.

> Até aqui, vimos destrutores não-virtuais, comuns: quando um objeto da classe **derivada** for destruído explicitamente (delete) a partir de um ponteiro da classe **base** para o objeto, C++ especifica que o comportamento é indefinido (código inseguro).

A solução simples é criar um **destrutor virtual** para a classe base. Isso torna virtuais todos os destrutores das classes derivadas, embora tenham outro nome (obviamente). Desta maneira, se um objeto da hierarquia for destruído explicitamente aplicando-se *delete* a um ponteiro da classe base, o destrutor para a classe apropriada é chamado de acordo com o objeto apontado.

Destrutores Virtuais

A regra é a seguinte:

Sempre crie um destrutor virtual caso sua classe tenha funções virtuais. Isso garante que um destrutor personalizado da classe derivada (se houver um) será chamado quando o objeto da classe derivada for excluído por meio de um ponteiro da classe base.

```
class Forma {
    protected:
        double x, y;
    public:
        Forma(double xx=0, double yy=0) : x{xx}, y{yy} {}
        virtual ~Forma() {}

        // Funções virtuais puras
        virtual void imprime_dados() const = 0;
        virtual double area() const = 0;
        virtual double perimetro() const = 0;
};
```

Referências

- https://cplusplus.com/reference/

- Material de aula ECOP03