AED - Algoritmos e Estruturas de Dados

Aula 4

Prof. Rodrigo Mafort

Tipo de Dados

- Até agora, usamos apenas tipos já existentes na linguagem
  - `int, float, char, double, long` …
- Mas existem casos em que um único tipo é insuficiente para representar a informação
- Por exemplo:
  - O registro de um carro: Ele é formado por placa, renavam, chassi, cor, modelo, fabricante...
- Como armazenar todas essas informações? Várias variáveis?

Definição de Tipos

- As linguagens de programação estruturadas permitem a definição de novos tipos de dados

- Esses novos tipos são **compostos** por vários campos de tipos de dados previamente definidos (int, float, ...)

- Um novo tipo também pode ser formado por outros tipos já definidos (Um tipo de dados para armazenar os dados de uma pessoa, pode conter um tipo que armazena a data de nascimento)

Exemplos

- Uma data:
  - Três campos do tipo inteiro (dia, mês e ano)

- As **coordenadas** de um ponto no plano:
  - Formada por dois campos do tipo real (X e Y)

- Os dados de um círculo:
  - As **coordenadas** do centro
  - Seu raio

Novos tipos de dados

- A definição de novos tipos de dados apresenta alguns nomes diferentes na literatura
  - Estruturas (de dados)
  - Tipos
  - Registros
  - **Structs**
  - Tipo Abstrato de Dados (TAD)

Definição de novos tipos em C

- Em C, os tipos são chamados de **structs**

- Sintaxe:

```
struct <nome do novo tipo> {
    <tipo já existente> <nome do campo>;
    <tipo já existente> <nome do campo>;
    ....
} <opcional: nome da variáveis>;
```

Exemplo

- Tipo para armazenar coordenadas no plano

```
struct coordenada {
    float X;
    float Y;
};
```

Utilização dos Tipos de Dados

- Para definir variáveis de um tipo:
  - Opção 1: Criar as variáveis durante a definição do tipo:

```
struct coordenada {
    float X;
    float Y;
} A, B;
```

As variáveis A e B já são definidas no momento da construção do tipo **coordenada**.

Utilização dos Tipos de Dados

- Para definir variáveis de um tipo:
  - Opção 2: Criar as variáveis após a definição do tipo (em qualquer momento):

```
struct coordenada {
    float X;
    float Y;
};
...
struct coordenada A, B;
```

Atenção!
Essa opção exige que se comece a definição com a palavra **struct**:
Defina duas variáveis A e B do tipo **estrutura coordenada**

Acesso aos dados dentro do tipo

```
struct coordenada {
      float X;
      float Y;
} A, B;
```

- Como acessar o valor de X **dentro** de cada coordenada?
  - Operador . (ponto): <variável do tipo>.<campo>
  - Ex:

```
A.X = 10.0;
if (A.X == B.X && A.Y == B.Y)
      printf("Os pontos A e B são iguais\n");
```

Exemplo: Definição dos Structs

```
struct coordenada {
        float X;
        float Y;
};
struct circulo {
        struct coordenada centro;
        float raio;
};
```

O centro do círculo também é uma estrutura (struct coordenada)

```
struct circulo C;
printf("Digite os dados do circulo\n");
scanf("%f %f %f", &C.centro.X, &C.centro.Y, &C.raio);

struct coordenadas P;
printf("Digite as coordenadas do Ponto\n");
scanf("%f %f", &P.X, &P.Y);
```

Comando typedef

- Permite criar um “apelido” para os tipos definidos

Sem typedef:

```
struct pessoa {
   int idade;
   float peso;
   float altura;
};
struct pessoa Joao;
struct pessoa Maria;
```

Com typedef:

```
typedef struct {
   int idade;
   float peso;
   float altura;
} pessoa;
pessoa Joao;
pessoa Maria;
```

Comando typedef

- Permite criar um “apelido” para os tipos definidos

Com typedef:

```
typedef struct {
   int idade;
   float peso;
   float altura;
} pessoa;
pessoa Joao;
pessoa Maria;
```

- Observe que o tipo "`struct pessoa`" foi batizado com o nome "`pessoa`";
- Isso possibilita se referir ao nome escolhido como um novo tipo (tal como `int`, `float`, ...)
- **Importante:** Observe que "`pessoa`" não é uma variável. "`pessoa`" é **um novo tipo de dados**.

Exemplo: Definição dos Structs

```
typedef struct {
        float X;
        float Y;
} coordenada;
typedef struct {
        coordenada centro;
        float raio;
} circulo;
```

Com o typedef coordenadas passa a ser reconhecido como um outro tipo de dados (assim como int, float, char...)

E agora podemos usar como qualquer outro tipo de dados

Inclusive como um campo dentro de outra estrutura

```
circulo C;
printf("Digite os dados do circulo\n");
scanf("%f %f %f", &C.centro.X, &C.centro.Y, &C.raio);

coordenada P;
printf("Digite as coordenadas do Ponto\n");
scanf("%f %f", &P.X, P.Y);
```

Exercício

1. Escreva um programa que leia os dados de quatro pontos no plano. Calcule a área do retângulo formado por esses pontos.

Exercício

2. Um curso de inglês tem turmas com cinco alunos apenas. Cada aluno da turma é identificado por uma matrícula e por sua idade. Para ser aprovado em um determinado período, cada aluno deve ter média maior do que 6.0 e pelo menos duas notas maiores ou iguais a 7.0 dentre as cinco provas que ele faz durante o semestre. Escreva um programa que leia os alunos de uma turma, suas notas e imprima as matrículas dos alunos aprovados e, em seguida, dos reprovados. Faça a questão sem usar registros e usando registros.

Ponteiros para estruturas (struct)

```
typedef struct {
    long long int CPF;
    int idade;
    char nome[200];
} pessoa;
```

```
Pessoa p;
p.CPF = 12345678909
p.idade = 99;
p.nome = "Fulano"
```

```
pessoa *ptP = &p;
```

```
//Para acessar os campos de p:
*ptP.CPF: Quem é o ponteiro?
ptP ou CPF???
```

Para evitar essa confusão, o uso de ponteiros para estruturas tem um operador diferente:

Operador `->`

Ponteiros para estruturas (struct)

```
typedef struct {
    long long int CPF;
    int idade;
    char nome[200];
} pessoa;

Pessoa p;
p.CPF = 12345678909
p.idade = 99;
p.nome = "Fulano"
```

```
pessoa *ptP = &p

//Para acessar os campos de p:
ptP->idade = ptP->idade + 1;

//Não é necessário usar o *
antes de ptP. O uso de -> já
indica que é um ponteiro.

<ponteiro> -> <campo do struct>
```

Operador ->

- Permite acessar os campos de um ponteiro para estruturas

- Permite enviar uma estrutura como parâmetro (por referência)

<ponteiro para struct> -> <campo do struct>

Passagem de Structs por Parâmetro

- **Passagem por Valor:**

```
#include <stdio.h>
typedef struct
{
    float X;
    float Y;
} Coordenada;
void Imprimir(Coordenada c)
{
    printf("(%f, %f)\n",c.X,c.Y);
}
int main()
{
    Coordenada c;
    c.X = 4;
    c.Y = 3;
    Imprimir(c);
    return 0;
}
```

- **Passagem por Referência:**

```
void Deslocar(Coordenada *c,
              float dX, float dY)
{
    c -> X = c -> X + dX;
    c -> Y = c -> Y + dY;
}

int main()
{
    Coordenada c;
    c.X = 4;
    c.Y = 3;
    Deslocar(&c,5,5);
    Imprimir(c); //9 e 8
    return 0;
}
```

Dados

Carro1
Placa = AAA-1234
Modelo = Audi R8
Piloto = Zé Carioca
AirBag = Ativado

...

Moto5
Placa = BBB-6789
Modelo = BMW HP4
Piloto = Pato Donald

O sistema é responsável por associar os dados as funções adequadas.

Os dados e as funções são do programa.

Mas e agora?

Temos duas funções acelerar? Qual é para carros e qual é para motos?

Funções

Acelerar
Frear
VirarEsquerda
VirarDireita
AcenderFarol

Acelerar
Frear
VirarEsquerda
VirarDireita

Structs vs Classes

- Na programação estruturada, ao usar structs e funções/procedimentos existe uma separação entre os dados (variáveis) e as ações que devem ser executadas considerando esses dados (as funções e os procedimentos).

- Cabe ao programador passar a variável certa para o método adequado.

- Os dados e as funções não estão agrupados como um item lógico no programa.

Structs vs Classes

- Ao usar o conceito da programação orientada a objetos, os dados e as funções são agrupados como um único modelo lógico: classes e objetos (instâncias das classes).

- Durante a definição do programa, os dados (os atributos) e as funções (os métodos) são declarados em conjunto.

- Os dados e os métodos “andam” juntos, logo não é mais tarefa do programador inferir qual é a variável certa para cada método.

Structs vs Classes

- A “entidade” que agrupa os dados e os métodos é chamada de classe. Uma classe é um modelo a partir do qual são instanciados os objetos.

- Os objetos armazenam então:
  - Os dados, chamados de atributos
  - As funções e os procedimentos, chamados de métodos

  - Os métodos são as formas que o programador definiu para interagir com os objetos e seus atributos.

Structs vs Classes

- Ao “empacotar” dados (atributos) e ações (métodos) a POO permitiu o desenvolvimento de novos conceitos:
  - Encapsulamento: permite proteger determinadas informações de acesso indevido por parte do próprio programador. Determinados atributos só podem ser alterados através de métodos específicos.
  - Herança: a habilidade de reaproveitar classes já declaradas, definindo apenas novos atributos e métodos atrelados à nova classe (o inimigo do copiar e colar de código).
  - Polimorfismo: a habilidade de um objeto ser tratado como instância de uma outra classe ancestral.

Exemplo – Visto no Laboratório

1. Crie uma classe em C++ que represente uma coordenada no plano (X,Y). Implemente:
   a) A classe
   b) O construtor
   c) Um método que calcula a distância da coordenada atual até uma outra coordenada qualquer.

- Vamos analisar como esse exemplo difere considerando a implementação em programação estruturada da implementação em programação orientada a objetos.

Exercícios

1. Modele e implemente um cofrinho em C++:

   Considere que serão depositadas apenas moedas de 5, 10, 25 e 50 centavos e moedas de 1 real.

   Escreva métodos:

   - Para depositar as moedas
   - Para calcular o total depositado no cofrinho
   - Para sacar um valor do cofrinho
   - Para verificar se uma moeda é falsa (considere que moedas com valores diferentes dos descritos é considerada falsa)

   Atenção: Não é permitido alterar o número de moedas nem o valor do cofrinho sem usar os métodos descritos. Ao depositar, confira primeiro se a moeda não é falsa. Caso seja falsa, dispare uma exceção.

Ponteiros para Objetos

- Assim como é possível definir ponteiros para inteiros, reais, etc.., podemos definir ponteiros para objetos.

- Por exemplo:

```
Coordenada c1(5,10);
Coordenada* c2 = &c1;
```

- c2 nesse exemplo é um ponteiro que aponta para o objeto c1.
- Alterações em c2 na verdade estão modificando c1.

Ponteiros para Objetos

- Outro uso de ponteiros para objetos é a possibilidade de controlar a alocação/liberação de objetos.

- Para declarar um objeto:
  - `Coordenada* cA = new Coordenada(); //const. sem parâmetros`
  - `Coordenada* cB = new Coordenada(5,0);`

- Para liberar um objeto:
  - `delete cA;`
  - `delete cB;`

Ponteiros para Objetos

- Ao definir ponteiros para objetos (como no slide anterior), o acesso aos métodos e atributos deve ser feito através do operador ->.
- Esse operador evita a dúvida se o ponteiro é o objeto ou o atributo do objeto.

```
Coordenada* cA = new Coordenada();
cout << cA->x << " " << cA->y << endl;
delete cA;
```

Ponteiros para Objetos

- Vale observar que ao optar pelo uso dos ponteiros, o método definido na aula anterior para calcular distância entre dois pontos (que recebia um objeto da classe Coordenada) precisaria ser modificado ou duplicado.

- Uma versão receberia objetos da classe Coordenada
- A outra versão receberia ponteiros para objetos dessa classe

Ponteiros para Objetos

- A utilização dos ponteiros apresenta algumas vantagens:
  - Evita a realização de cópias dos objetos: todo o acesso será sempre realizado considerando a instância “original.
  - Permite a possibilidade de tratar um objeto de uma classe como um objeto de uma classe ancestral (requer os conceitos de herança e polimorfismo).
  - Possibilita o controle manual da alocação de memória (quando instanciar o objeto e quando liberar a memória que o objeto ocupa).

- Entretanto, existem alguns pontos que precisam de atenção:
  - Tal como na passagem por referência, alterações no objeto aponta são refletidas no “objeto original”.
  - O controle manual da memória implica que cabe ao programador verificar o momento adequado para liberar a memória alocada. Erros nesse processo podem levar a estouros de memória.

Contêineres C++

- Diferente do C, o C++ já apresenta muitos tipos de dados previamente definidos.

- Por exemplo:
  - array
  - vector
  - deque
  - set
  - list (lista duplamente encadeada)
  - map
  - queue (fila)
  - stack (pilha)

Contêineres C++

- Cada um desses contêineres apresentam características interessantes (como vimos ao longo do curso de ED)
- E cada um está definido em uma biblioteca que precisamos importar.

Contêineres C++

```
#include <stack>
#include <iostream>

using namespace std;

int main()
{
    stack<int> P;
    for (int i = 5; i >= 1; i--)
        P.push(i);
    while (P.empty() == false)
    {
        cout << P.top() << endl;
        P.pop();
    }
    return 0;
}
```

Dúvidas?

Pilares da Orientação a Objetos

- Existem quatro pilares básicos em que a orientação a objetos se apoia:
  - Abstração
  - Encapsulamento
  - **Herança**
  - Polimorfismo

- Na aula de hoje vamos ver os conceitos da Herança

Herança

- A programação orientada a objetos tem como princípio facilitar o desenvolvimento dos programas e sistemas.
- Uma das formas de fazer isso é evitar que uma parte do código tenha quer ser copiada entre diferentes classes.
- Por exemplo:

Paciente
Nome CPF Data de Nascimento Plano de Saúde Consulta Agendada Próxima Consulta
Agendar Consulta ....

Médico
Nome CPF Data de Nascimento Especialidade CRM
Preparar Atestado ....

Paciente e médico tem características em comum!

Ambos são pessoas...

Vamos precisar implementar cada atributo duas vezes...

Herança

- No exemplo, tanto o médico quanto o paciente são pessoas...

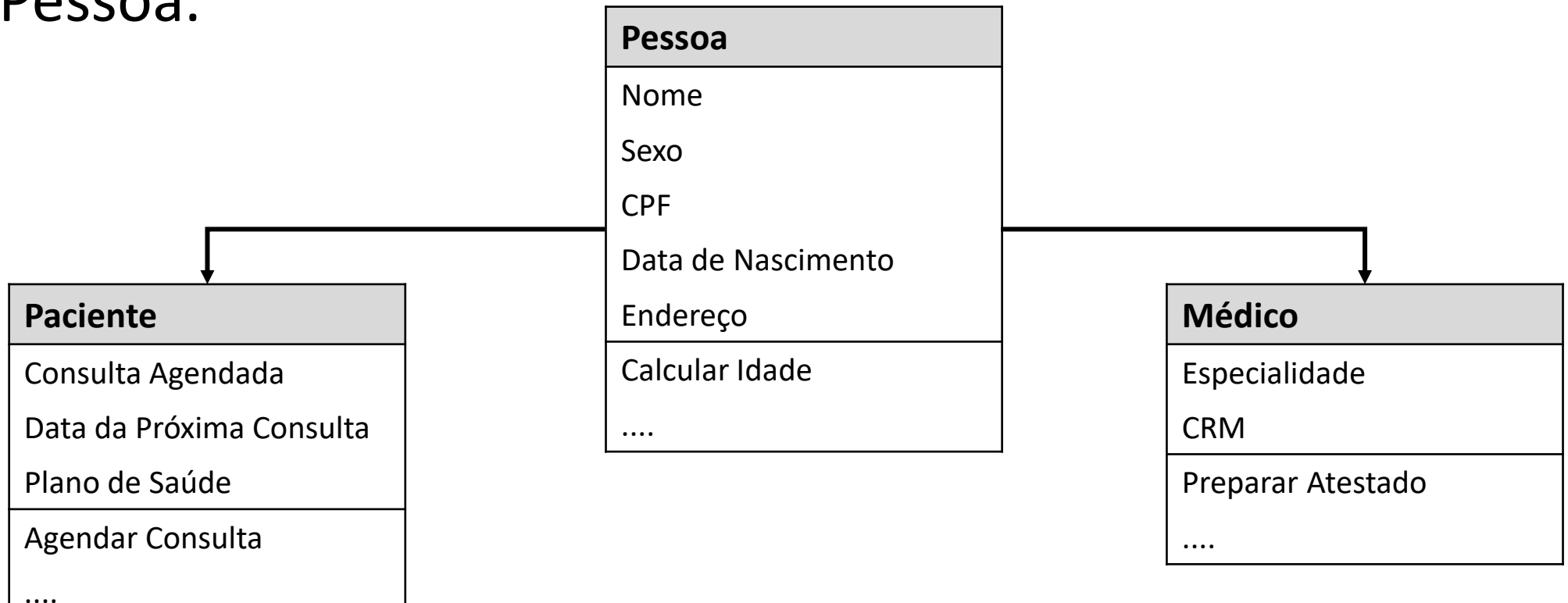
- E se pudéssemos modelar as pessoas?
- Vamos abstrair os detalhes dos médicos e dos pacientes

Pessoa
Nome Sexo CPF Data de Nascimento Endereço
Calcular Idade ....

- Agora precisamos "reusar" esse modelo... Como podemos fazer isso?

Herança

- A ideia da herança é permitir que uma classe HERDE os atributos e métodos de uma outra.
- No exemplo: Paciente e Médico herdam os atributos e métodos de Pessoa.



Herança

- Quando definimos classes com características (atributos e métodos) idênticos a outras classes, podemos reaproveitar essa parte do código já escrita. As duas classes passam a herdar essas características.

- A nova classe herda as características da outra classe
- E a estende, adicionando as características específicas dessa nova classe.

- A nova classe é chamada de classe **filha** ou **subclasse**
- A classe que foi herdada é chamada de **classe mãe**, **superclasse** ou **base**.

Herança em C++

- Como a ideia da herança é permitir que uma classe herde as características de uma outra, precisamos começar implementando a classe mãe.

- No nosso exemplo, vamos implementar primeiro a classe Pessoa.

Pessoa
Nome Sexo CPF Data de Nascimento Endereço
Calcular Idade ....

Classe Pessoa

Exemplo completo no GitHub

```
class Pessoa
{
    public:
        Pessoa(string _nome, string _sexo, string _cpf, Data _nascimento,
                                                        string _endereco);

        virtual ~Pessoa();

        string getNome();
        string getSexo();
        string getCPF();
        Data getNascimento();
        string getEndereco();
        int GetIdade();
```

Classe Pessoa

Exemplo completo no GitHub

```
class Pessoa
{
    //... Parte public: -- Slide anterior

    protected:
        string nome;
        string sexo;
        string cpf;
        Data dataNascimento;
        string endereco;

    private:
};
```

Herança em C++

- Agora que definimos a classe base, podemos reusar esse código criando as classes filhas.

- No nosso exemplo, médico e paciente

- Médico herda de Pessoa
- Paciente herda de Pessoa

- As duas classes serão subclasses da classe Pessoa

Herança em C++

- Para definir que uma classe herda as características de uma outra, precisamos deixar isso claro na definição.

- Para isso, ao batizar a classe, precisamos dizer de quem ela é filha.

class <nome da classe filha> : <modificador> <nome da classe mãe>

- Onde modificador pode ser `public, protected ou private`
- Mais comum: uso de herança do tipo `public`

Herança em C++

- Os modificadores aplicados a herança permitem alterar o nível de encapsulamento dos atributos e métodos que serão herdados das classes ancestrais.
- Modificador Public: Mantem o nível de encapsulamento original da classe ancestral
- Modificador Protected: Atributos públicos da classe ancestral são transformados em atributos protegidos.
- Modificador Private: Atributos públicos e protegidos da classe ancestral são transformados em atributos privados. A classe filha não acessa nenhuma informação da classe ancestral.

Herança em C++

Encapsulamento dos Atributos e Métodos	Modificador de Herança		
	Public	**Protected**	**Private**
Public	• Permanecem públicos na classe filha. • Acesso externo e interno	• São transformados em atributos/métodos protegidos na classe filha. • Acesso apenas interno.	• São transformados em atributos/métodos privados na classe filha. • Acesso por meio de métodos públicos e protegidos, se existirem.
Protected	• Permanecem protegidos na classe filha • Acesso somente interno	• Permanecem protegidos na classe filha • Acesso somente interno	• Idem anterior.
Private	• Permanecem privados na classe filha • Filha só acessa por meio de métodos públicos e protegidos, se existirem.	• Permanecem privados na classe filha • Filha só acessa por meio de métodos públicos e protegidos, se existirem.	• Idem anterior.

A Classe Filha: Paciente

```
class Paciente : public Pessoa
{
    public:
        Paciente(string _nome, string _sexo, string _cpf, Data _nascimento,
                                                    string _endereco, string _plano);

        virtual ~Paciente();
        void MarcarConsulta(Data _data);
        void CancelarConsulta();
        void RemarcarConsulta(Data _data);
        string getPlanoSaude();
        void setPlanoSaude(string _plano);

    protected:
        Data proximaConsulta;
        bool consultaAgendada;
        string planoDeSaude;
};
```

Herança: Paciente tem todos os atributos e métodos de Pessoa

Especialização: E adiciona seus próprios métodos e atributos.

A Classe Filha: Médico

```
class Medico : public Pessoa
{
    public:
        Medico(string _nome, string _sexo, string _cpf, Data _nascimento,
                    string _endereco, string _crm, string _especialidade);
        virtual ~Medico();
        string GetCRM();
        string GetEspecialidade();
    protected:
        string crm;
        string especialidade;
        void SetCRM(string _crm);
        void SetEspecialidade(string _especialidade);
};
```

E os construtores e destrutores?

- Quando definimos a classe base, implementamos também seus construtores e destrutores.
- Os construtores das classes ancestrais podem inicializar atributos privados que não podemos acessar diretamente na classe filha.
- Desta forma, precisamos de uma forma de disparar esses métodos a partir das classes filhas.
  - Construtor da classe filha chama construtores das classes ancestrais
  - Destrutor da classe filha chama destrutores das classes ancestrais

Como chamar o construtor da classe mãe

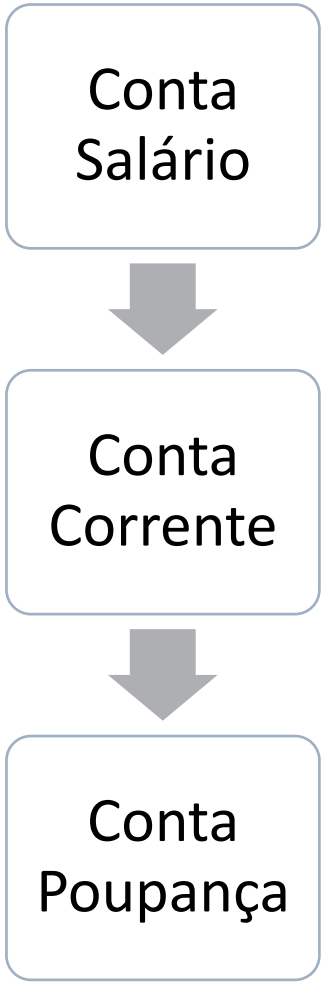
```
<classe filha>::<construtor_filha(<parâmetros>)
: <construtor_mãe>(<parâmetros>)
{
    //Inicializar os atributos da classe filha
}
```

Como chamar o construtor da classe mãe

```
Medico::Medico(string _nome, string _sexo,
string _cpf, Data _nascimento, string _endereco,
string _crm, string _especialidade)
: Pessoa(_nome, _sexo, _cpf, _nascimento,
                                    _endereco)
{
    //construir atributos da classe filha
    SetCRM(_crm);
    SetEspecialidade(_especialidade);
}
```

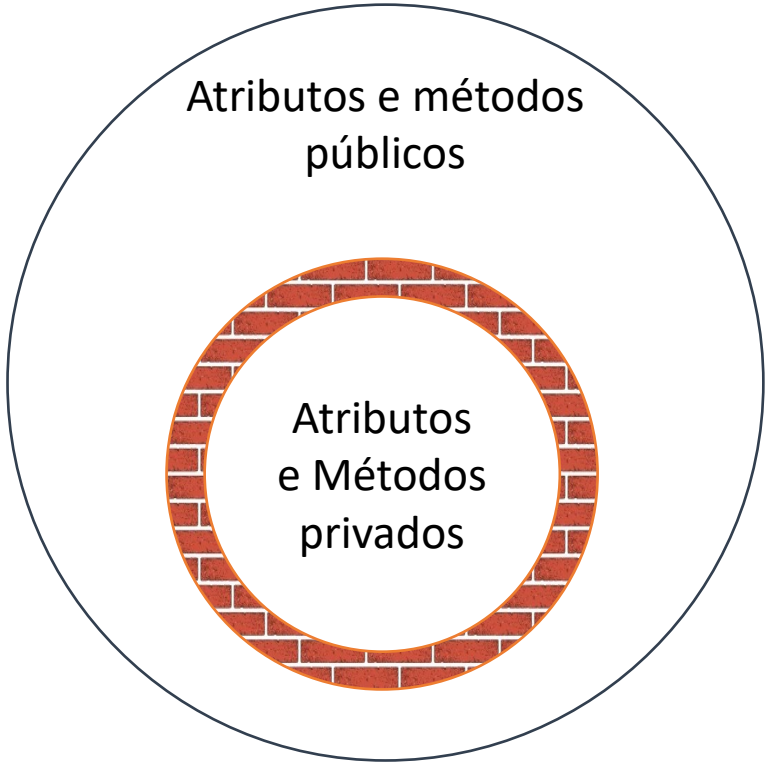
Herança – Árvore de Herança

- Em alguns casos, podemos implementar uma verdadeira árvore de herança entre classes:
- Conta Salário
- Conta Corrente herda de Conta Salário
- Conta Poupança herda de Conta Corrente
- Essa implementação garante que conta poupança terá todos os atributos e métodos da conta salário.



Encapsulamento e Herança

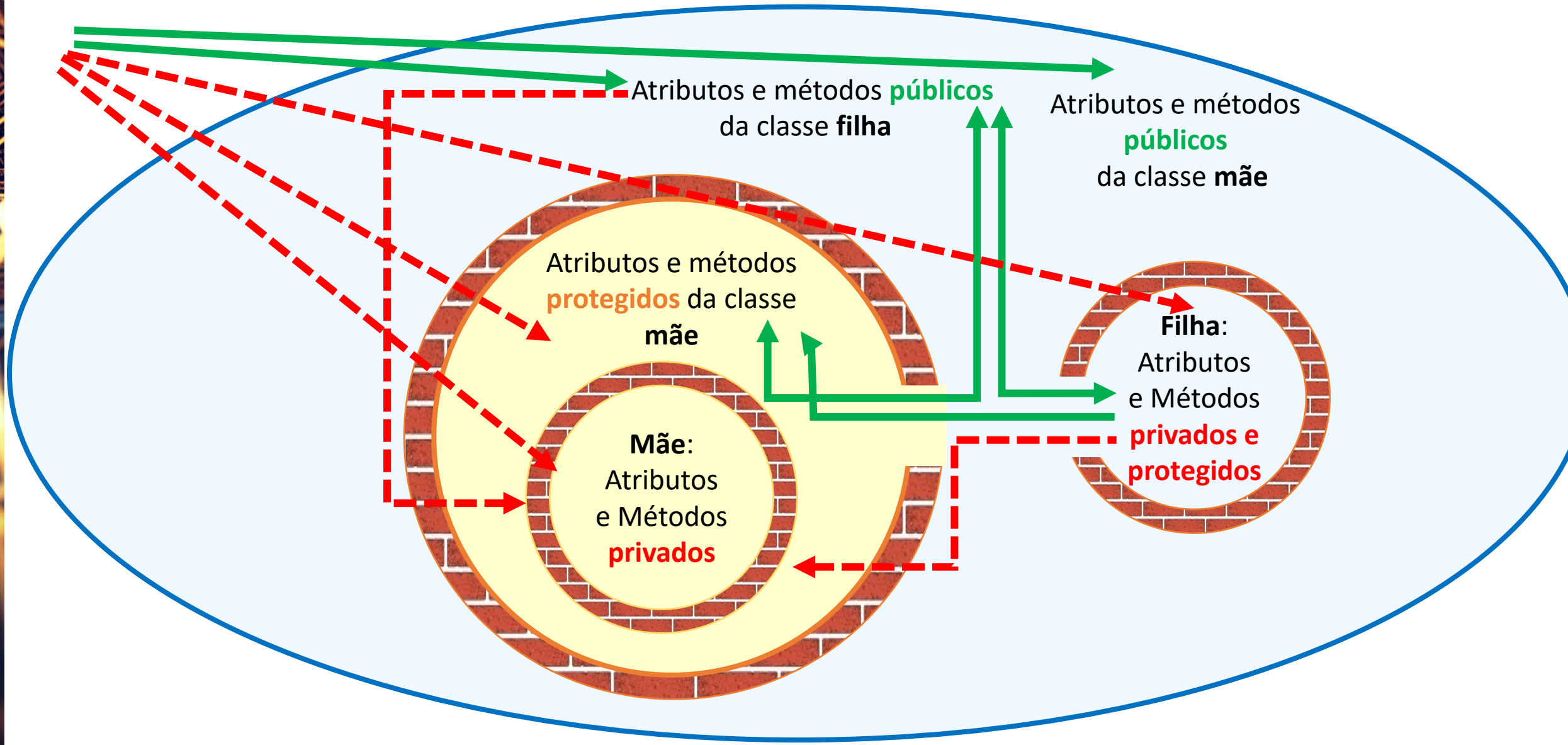
- Como vimos anteriormente, ao definir uma classe temos dois níveis de proteção aos atributos: públicos e privados.



- Como essa proteção se comporta quando essa classe se torna ancestral de uma outra classe?

- Os métodos e atributos privados da classe mãe, continuam privados!
- Até para as classes filhas...

Encapsulamento e Herança



Encapsulamento e Herança

- Classe mãe pode acessar:
  - Atributos e métodos públicos da classe mãe
  - Atributos e métodos protegidos da classe mãe
  - Atributos e métodos privados da classe mãe

- Classe filha pode acessar:
  - Atributos e métodos públicos da classe filha
  - Atributos e métodos protegidos da classe filha
  - Atributos e métodos privados da classe filha
  - Atributos e métodos públicos da classe mãe
  - Atributos e métodos protegidos da classe mãe

- Por que a classe filha não pode acessar os atributos privados da mãe?
  - Porque são privados da mãe... Somente ela tem acesso (lembrar privado ≠ protegido)

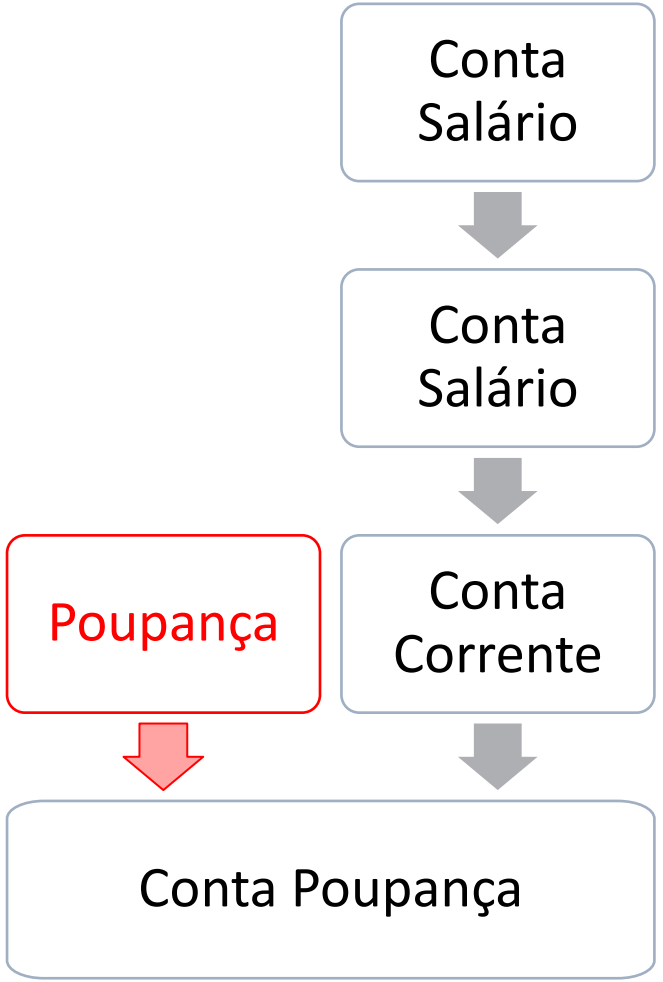
- Por que a classe mãe não pode acessar os atributos da filha?
  - Porque a mãe não herda da filha... A relação é ao contrário: a filha herda da mãe.

Herança Múltipla

- No exemplo das contas, o que aconteceria se tivéssemos uma classe para poupança?

- Conta Salário
- Conta Corrente herda de Conta Salário
- Conta Poupança herda de Conta Corrente **e de Poupança**

- Esse conceito é chamado de Herança Múltipla e será assunto da próxima aula



Atividade de Herança

Considere uma empresa de consultoria jurídica. Essa empresa tem clientes e funcionários.

Todo cliente tem uma ficha cadastral com seu nome, e-mail, endereço, telefone e um limite de consultas mensais.

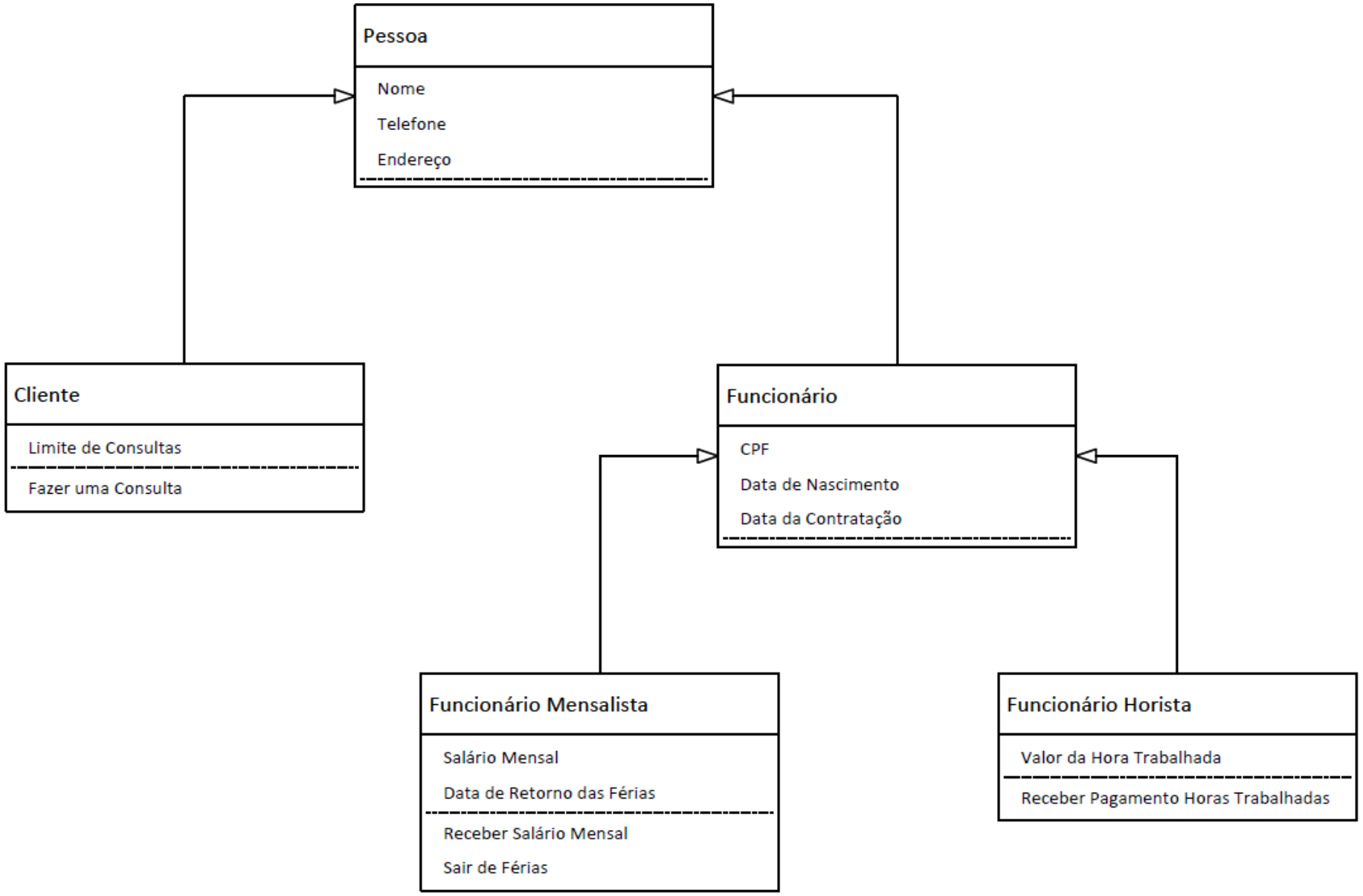
Os funcionários da empresa podem ser contratados de duas formas diferentes: mensalista ou horista. Independente do vínculo trabalhista, a empresa armazena o CPF, a data de nascimento e a data da contratação de cada funcionário.

Os funcionários mensalistas tem um salário fixo mensal e direito à férias anuais. Sempre que um funcionário mensalista sai de férias, a data de retorno é armazenada para controle.

Os funcionários horistas tem um salário baseado no número de horas trabalhadas. Desta forma, seu cadastro deve armazenar o valor fixo pago por hora de trabalho. Esses funcionários não tem direito a férias.

1. Conhecendo o modelo de trabalho, modele as classes e seus relacionamentos.
2. Em seguida, implemente seu modelo em C++.

Exemplo de Modelo



Métodos virtuais

- Vamos considerar o seguinte modelo:
  - Vamos implementar polígonos: quadrados, círculos e triângulos.
  - Para isso, vamos construir uma classe chamada de Polígono
  - Essa classe vai ser ancestral de todos os nossos polígonos

- Vamos considerar que todos as classes tem um método chamado Desenhar().
- Como esse método é comum a todos, vamos definir esse método na classe ancestral.

Métodos virtuais

- Quando um método é marcado como virtual estamos definindo que ele pode ser sobrescrito pela classe filha.

- No exemplo, polígono tem um método Desenhar()
- Só que não sabemos desenhar polígonos quaisquer.

- Mas ao definir as classes filhas de polígono, podemos sobrescrever esse método.
- Entretanto: todo polígono terá um método Desenhar()
- Isso se comporta como um contrato firmado durante a herança

Métodos e classes abstratas

- Quando definimos uma classe com métodos virtuais, muitas vezes não queremos que essa classe seja realmente instanciada (não queremos objetos dessa classe).
- Nesse caso, essa classe assume ainda mais esse aspecto de contrato: toda classe que herdar deverá cumprir o contrato, sobrescrevendo o métodos marcados como virtuais.
- Para indicar que um método é abstrato precisamos:
  1. Marcar como virtual
  2. Indicar no arquivo.h que esse método recebe 0.
- `virtual void Desenhar() = 0;`
- Importante: a existência de um método abstrato implica que todo a classe é abstrata.
- Desta forma, essa classe não poderá ser instanciada.

Observe o exemplo:

```
int main()
{
    Quadrado q;
    Circulo c;

    vector<Poligono*> vet;
    vet.push_back(&q);
    vet.push_back(&c);

    for (int i = 0; i < vet.size(); i++)
        vet[i]->Desenhar(); //O que será impresso?

    return 0;
}
```