Residência em Software

# De Estruturas e Funções para Orientação a Objetos


Professores:

Álvaro Coelho, Edgar Alexander, Esbel Valero e Hélder Almeida



INSTITUIÇÃO EXECUTORA: cepedi, UESC

COORDENADORA: MCTI FUTURO, Softex

APOIO: Ministério da Ciência, Tecnologia e Inovação, Governo Federal Brasil União e Reconstrução

Residência em Software

# Introdução e objetivo da aula

Objetivo: demonstrar a transição da programação estruturada para a orientação a objetos, baseando-se em structs e funções de C++ como base.

Residência em Software

# Structs e Funções

Projeto data… revisão de conceitos

```
5  struct Data {
6      int dia;
7      int mes;
8      int ano;
9  };
10
11 bool dataValida(int dia, int mes, int ano) {
12     if (ano < 1900 || ano > 2100) { return false; }
13     if (mes < 1 || mes > 12) { return false; }
14     int diasNoMes = 0;
15     if (mes == 2) {
16         diasNoMes =
17         (ano % 4 == 0 && (ano % 100 != 0 || ano % 400 == 0)) ? 29 : 28;
18     } else if (mes == 4 || mes == 6 || mes == 9 || mes == 11) {
19         diasNoMes = 30;
20     } else {
21         diasNoMes = 31;
22     }
23     if (dia < 1 || dia > diasNoMes) { return false; }
24
25     return true;
26 }
```

```
29 int main(){
30     struct Data data;
31     data.dia = 15;
32     data.mes = 8;
33     data.ano = 2023;
34
35     if(dataValida(data.dia,data.mes,data.ano)){
36         cout << data.dia << "/";
37         cout << data.mes << "/";
38         cout << data.ano << endl;
39     }
40     else{
41         cout << "Data inválida";
42     }
43
44     return 0;
45 }
```

Residência em Software

# Reflexão

Quais possíveis melhorias poderiam ser feitas na organização deste código?

- função “dataValida” receber a struct?
- criar um inicializador para a struct?
- verificar se a data é válida na inicialização?
- como fazer isso?

Residência em Software

# Modularização de Código

A modularização na organização do código oferece as seguintes vantagens:

- **Reutilização de Código**: Módulos podem ser reutilizados em diferentes partes do programa.
- **Facilidade de Manutenção**: Cada módulo é uma unidade separada, facilitando a localização e correção de erros.
- **Organização Lógica**: O código é dividido em partes menores, tornando-o mais claro e lógico.
- **Colaboração**: Diferentes desenvolvedores podem trabalhar em módulos separados simultaneamente.
- **Escalabilidade**: Adicionar novos recursos é mais fácil, pois não afeta todo o código.
- **Abstração**: Os detalhes internos de um módulo podem ser ocultados, tornando o código mais compreensível.
- **Testabilidade**: Módulos individuais podem ser testados de forma isolada, facilitando a depuração.

A modularização contribui para um código mais organizado, eficiente e fácil de manter.

# Structs e Funções

Melhorando o exemplo anterior

```
28  bool dataValida(struct Data data){
29      return dataValida(data.dia,data.mes,data.ano);
30  }
31
32  struct Data inicializaData(int dia, int mes, int ano){
33      struct Data data;
34
35      if(dataValida(dia,mes,ano)){
36          data.dia = dia;
37          data.mes = mes;
38          data.ano = ano;
39      }
40      else{
41          cout << "Data inválida, inicializando em 1/1/1900" << endl;
42          data.dia = 1;
43          data.mes = 1;
44          data.ano = 1900;
45      }
46
47      return data;
48  }
```

```
50  int main(){
51      struct Data data = inicializaData(33,8,2023);
52
53      cout << data.dia << "/";
54      cout << data.mes << "/";
55      cout << data.ano << endl;
56
57
58      return 0;
59  }
```

Residência em Software

# Mais reflexão…

Ainda poderia melhorar?

- função para imprimir a data?
- a data pode ser impressa de várias formas, dd/mm/yyy ou mm/dd/yyyy ou yyyy/mm/dd… como resolver?
- essa função de inicialização cria uma cópia da struct. Como melhorar?
- como alterar a data depois de inicializada?

Residência em Software

# Structs como Tipos de Dados Abstratos

Tipos de Dados Abstratos (**TDAs**) são estruturas de dados que **encapsulam dados** e as **operações** que podem ser realizadas nesses dados. Eles fornecem uma **abstração** de alto nível que permite ao programador usar esses tipos sem precisar conhecer detalhes de implementação subjacentes. TDAs são uma maneira de **ocultar a complexidade** e promover a **modularização** do código, facilitando o desenvolvimento, a manutenção e a compreensão do software.

Exemplos comuns de TDAs incluem pilhas, filas, listas vinculadas, árvores e grafos.

Residência em Software

# Organizando o Código com Structs e Funções

Podemos combinar structs e funções para criar programas organizados.

Utilizar structs como parâmetros de funções para manipulação de seus valores.

Modificador de structs em funções pode ser um problema que é solucionado em Orientação a Objetos com escopo definido nos métodos

Residência em Software

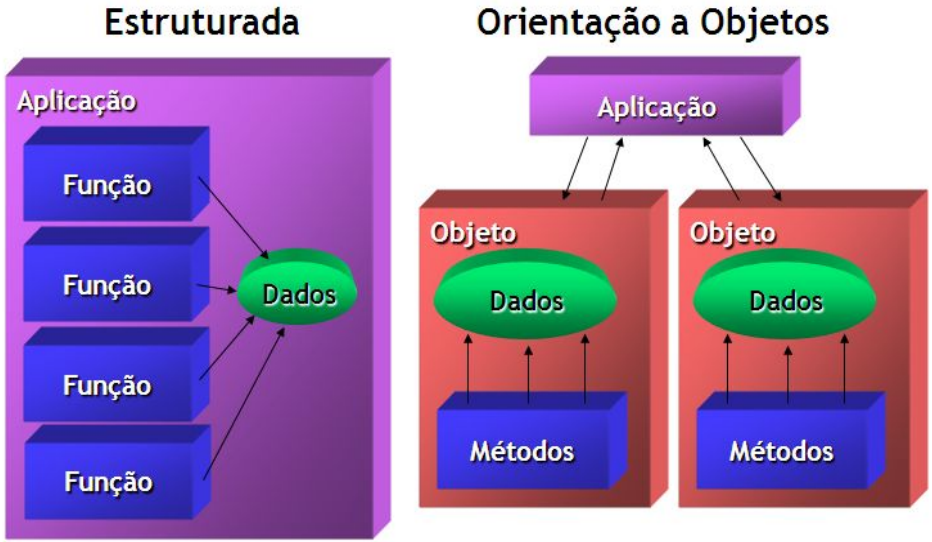
# Mais uma versão de nosso projeto

```
43  void alteraData( struct Data *data, int dia, int mes, int ano){
44      if (dataValida(dia,mes,ano)){
45          data->dia = dia;
46          data->mes = mes;
47          data->ano = ano;
48      }
49      else{
50          cout << "Data inválida, atribuindo 1/1/1900" << endl;
51          data->dia = 1;
52          data->mes = 1;
53          data->ano = 1900;
54      }
55
56  }
57
58  string to_string_zeros(int numero){
59      string numeroString = to_string(numero);
60
61      if(numero < 10){
62          numeroString = "0" + numeroString;
63      }
64
65      return numeroString;
66  }
```

```
68  string dataParaString(struct Data data,string format = "pt-br"){
69
70      string dia = to_string_zeros(data.dia);
71      string mes = to_string_zeros(data.mes);
72      string ano = to_string(data.ano);
73
74      if(format == "iso-8601")
75          return  ano+"/"+mes+"/"+dia;
76      else
77      if(format == "en-us")
78          return  mes+"/"+dia+"/"+ano;
79      else
80          return dia+"/"+mes+"/"+ano;
81  }
82
83
84  int main(){
85      struct Data data;
86
87      alteraData(&data,15,8,2023);
88      cout << dataParaString(data,"en-us") << endl;
89      cout << dataParaString(data) << endl;
90
91      alteraData(&data,30,2,2024);
92      cout << dataParaString(data,"en-us") << endl;
93      cout << dataParaString(data) << endl;
94
95      return 0;
96  }
```

# Limitações da Programação Estruturada

1. Falta de **abstração** e **organização**;
2. Complexidade para implementar **modularidade**;
3. Dificuldades em **escalabilidade**;
4. **Reutilização** limitada de código;
5. Dificuldade na **manutenção**;
6. Dificuldade na **colaboração;**
7. Limitações na **herança**;
8. Dificuldades em **modelagens complexas**.



Residência em Software

# Falta de Abstração e Organização

A programação estruturada não oferece uma maneira natural de representar objetos do mundo real com todas as suas características e comportamentos. Há falta de organização e separação clara das responsabilidades. Na POO, os objetos podem ser modelados de forma mais abstrata e representar entidades do mundo real.

**Programação Estruturada**: Ao representar um banco, pode ser difícil abstrair os diferentes tipos de contas (poupança, corrente, investimento) com todas as suas características únicas usando apenas funções e estruturas de dados.

**Programação Orientada a Objetos**: Na POO, você pode criar classes para representar cada tipo de conta, encapsulando suas características e comportamentos específicos.

Residência em Software

# Complexidade para implementar modularidade;

Em programação estruturada, a modularização é feita usando funções, mas essas funções não podem encapsular dados relacionados. Na POO, as classes permitem encapsular tanto dados quanto métodos que operam nesses dados, proporcionando maior modularidade.

**Programação Estruturada**: Em um sistema de gerenciamento de biblioteca, pode ser complicado manter dados sobre livros, usuários e empréstimos separados e independentes em um programa estruturado.

**Programação Orientada a Objetos**: Usando classes, você pode criar uma classe "Livro", uma classe "Usuário" e uma classe "Empréstimo", cada uma com seus próprios métodos e dados, proporcionando maior modularidade.

Residência em Software

# Dificuldades em Escalabilidade

À medida que um programa cresce em tamanho e complexidade, a programação estruturada pode se tornar difícil de gerenciar sua lógica e mantê-lo. Na POO, a estrutura hierárquica de classes permite escalabilidade mais eficiente.

**Programação Estruturada**: À medida que um sistema de pagamento online cresce, adicionar novos recursos e funcionalidades pode se tornar um desafio, pois as funções podem se tornar complexas e interdependentes.

**Programação Orientada a Objetos**: Usando a POO, você pode criar novas classes para cada novo recurso e estender a funcionalidade existente com mais facilidade, facilitando a escalabilidade.

Residência em Software

# Reutilização limitada de código

Em programação estruturada, a reutilização de código é limitada, uma vez que funções são independentes e não encapsulam dados. A POO facilita a reutilização, pois objetos e classes podem ser reutilizados em diferentes partes do programa.

**Programação Estruturada**: Em um sistema de processamento de pedidos, funções para calcular impostos e descontos podem ser usadas apenas localmente, tornando a reutilização difícil.

**Programação Orientada a Objetos**: Na POO, você pode criar classes para representar cálculos de impostos e descontos, tornando essas funcionalidades reutilizáveis em diferentes partes do programa.

Residência em Software

# Dificuldade de Manutenção

Alterações em um programa estruturado podem afetar outras partes inesperadamente, tornando a manutenção complexa. A POO, com seu princípio de encapsulamento, facilita a manutenção, pois as alterações em uma classe não afetam outras partes do programa.

**Programação Estruturada**: Em um sistema de gestão de inventário, fazer uma alteração em como os produtos são registrados pode afetar inadvertidamente a funcionalidade de outras partes do sistema.

**Programação Orientada a Objetos**: Com a POO, você pode encapsular a lógica de registro de produtos em uma classe específica, reduzindo o risco de impacto em outras partes do código ao fazer alterações.

Residência em Software

# Desafios na Colaboração entre Desenvolvedores

Em grandes projetos, a colaboração entre desenvolvedores pode ser desafiadora na programação estruturada, devido à falta de encapsulamento e separação clara de responsabilidades. A POO promove a colaboração mais eficaz, pois cada classe pode ser desenvolvida independentemente.

**Programação Estruturada**: Em um projeto de grande escala, diferentes programadores podem ter dificuldade em trabalhar em conjunto, pois as funções e variáveis globais podem causar conflitos.

**Programação Orientada a Objetos**: Com a POO, cada desenvolvedor pode trabalhar em classes específicas com suas próprias responsabilidades, reduzindo conflitos e facilitando a colaboração.

Residência em Software

# Limitações na herança

A programação estruturada não oferece um mecanismo natural para a herança de funcionalidades de uma parte do código para outra. A POO permite a herança, facilitando a criação de hierarquias de classes.

**Programação Estruturada**: Não há um mecanismo natural para herdar funcionalidades de uma função em outra parte do código.

**Programação Orientada a Objetos**: A POO permite criar classes base que podem ser herdadas por outras classes, permitindo a reutilização de código com herança.

Residência em Software

# Dificuldades em Modelagens Complexas

Modelar sistemas complexos e hierarquias de entidades do mundo real pode ser difícil na programação estruturada. A POO é mais adequada para representar sistemas complexos de forma intuitiva.

**Programação Estruturada**: Modelar um simulador de tráfego urbano com todas as entidades e interações pode ser complicado e desorganizado na programação estruturada.

**Programação Orientada a Objetos**: A POO permite modelar as diferentes entidades (carros, semáforos, pedestres) como objetos com seus próprios comportamentos, tornando a modelagem mais clara e eficiente.

Residência em Software

# Vamos praticar em grupo?

Problema: **Controle de Estoque *(utilizando structs e funções)***

*Você é o gerente de uma pequena loja de eletrônicos e precisa de um sistema de controle de inventário mais eficiente. Atualmente, você está gerenciando informações sobre produtos em estoque, incluindo nome do produto, quantidade em estoque e preço unitário, usando planilhas.*

Seu objetivo é criar um programa que permita o seguinte:

**Adicionar Produto**: Você deve ser capaz de adicionar novos produtos ao estoque, especificando o nome do produto, a quantidade em estoque e o preço unitário.

**Atualizar Estoque**: O programa deve permitir que você atualize a quantidade em estoque de um produto após cada venda.

**Calcular Valor Total**: Você deve ser capaz de calcular o valor total do estoque da loja, somando os valores de todos os produtos disponíveis.

**Listar Produtos**: O programa deve permitir a listagem de todos os produtos em estoque, incluindo seus nomes, quantidades e preços unitários.