

Structs

Tipo Abstrato de Dado (TAD)

Prof. Denio Duarte

duarte@uffs.edu.br

Prof. Caio V. Koch

caio.santos@uffs.edu.br

Structs

1. Até agora, vimos variáveis escalares e arranjos (matriz e vetor)
 - Todos armazenam tipos homogêneos
2. Propriedades importantes de um arranjo:
 - Todos os elementos são do mesmo tipo (homogêneo)
 - Para selecionar um elemento, devemos especificar sua posição
3. Usamos uma **struct** para armazenar uma coleção de dados de tipos possivelmente diferentes (heterogêneos)
4. Propriedades importantes de uma **struct**:
 - Os elementos (membros) de uma struct podem ser de tipos diferentes
 - Para selecionar um elemento de uma struct, devemos especificar o caminho até o elemento: **variavel_estrutura.elemento**

Structs - declaração de variáveis

1. Para declarar variáveis que são structs, podemos escrever

```
struct tdata {  
    int dia;  
    int mes;  
    int ano;  
};  
struct tdata data1, data2;
```

```
struct tfunc {  
    int id;  
    char nome[TAM_NOME+1];  
    double salario;  
};  
struct tfunc func1, func2;
```

2. Representação de data1 na memória do computador:



3. Os nomes dos membros de uma struct não conflitam com outros nomes de fora da struct

Structs - operações

1. Para acessar um membro de uma variável que é uma struct, devemos especificar o caminho até o elemento. Opções:
 - var_estrutura.elemento
 - var_estrutura->elemento (veremos mais tarde)

```
struct tdata {  
    int dia;  
    int mes;  
    int ano;  
};  
struct tdata data1, data2;
```

```
struct tfunc {  
    int id;  
    char nome[TAM_NOME+1];  
    double salario;  
};  
struct tfunc func1, func2;
```

```
printf("Dia: %d\n", data1.dia);  
printf("Nome do funcionario: %s\n", func1.nome);
```

Structs - operações

1. Podemos atribuir valores aos membros de uma variável que é uma struct e usá-los em operações aritméticas (quando cabível)

```
struct tdata {  
    int dia;  
    int mes;  
    int ano;  
};  
struct tdata data1, data2;
```

```
struct tfunc {  
    int id;  
    char nome[TAM_NOME+1];  
    double salario;  
};  
struct tfunc func1, func2;
```

```
data1.dia = 3;  
media = (func1.salario + func2.salario) / 2;  
  
scanf("%d", &data2.mes);  
scanf("%lf", &func1.salario);
```

Structs - operações

1. Diferentemente dos arranjos, podemos usar o operador = para atribuir uma struct a outra struct - desde que as structs sejam de tipos compatíveis

```
struct tdata {  
    int dia;  
    int mes;  
    int ano;  
};  
struct tdata data1, data2;
```

```
struct tfunc {  
    int id;  
    char nome[TAM_NOME+1];  
    double salario;  
};  
struct tfunc func1, func2;
```

```
data1 = data2;  
func2 = func1;
```

2. O efeito do comando `data1 = data2;` é copiar `data2.dia` para `data1.dia`, `data2.mes` para `data1.mes` e `data2.ano` para `data1.ano`.

Structs - nomeando tipos

1. Criar variáveis do tipo **struct**
2. Opção 1 (já vista): Definir como **struct nome nome_var;**

```
struct tdata {  
    int dia;  
    int mes;  
    int ano;  
};  
struct tdata data1, data2;
```

```
/* duas variáveis do tipo estrutura são criadas data1 e data2 */  
struct tdata data1, data2;
```

3. Nas declarações acima, não é possível omitir a palavra `struct`!

Structs - nomeando tipos

1. Criar variáveis do tipo **struct**
2. Opção 2: Definir um novo tipo usando `typedef`

```
typedef struct {  
    int dia;  
    int mes;  
    int ano;  
} Data;
```

```
typedef struct funcionario {  
    int id;  
    char nome[TAM_NOME+1];  
    double salario;  
} Funcionario;
```

```
Data data1, data2;  
Funcionario func1, func2;
```

```
struct tdata {  
    int dia;  
    int mes;  
    int ano;  
};  
typedef struct tdata Data;
```

Structs - como argumentos e retorno de funções

1. Funções podem receber structs como argumentos e retornar structs

```
void imprimeData(Data data) {  
    printf("Dia: %d\n", data.dia);  
    printf("Mes: %d\n", data.mes);  
    printf("Ano: %d\n", data.ano);  
}
```

```
Data constroiData(int dia, int mes, int ano) {  
    Data data;  
    data.dia = dia;  
    data.mes = mes;  
    data.ano = ano;  
    return data;  
}
```

```
imprimeData(data1);  
  
data2 = constroiData(9, 11, 2003);
```

Exercícios

1. Escreva as seguintes funções considerando o tipo `Data` definido nesta apresentação:
 - a. `int extraiDia(Data data)`
Retorna o dia que compõe a data passada como parâmetro (`data`).
 - b. `int comparaDatas(Data data1, Data data2)`
Retorna `-1` se a data `data1` é anterior à data `data2`, `1` se a data `data1` é posterior à data `data2` e `0` se as datas `data1` e `data2` são iguais.
2. Declare um tipo `Fraction` que consista em uma struct contendo dois membros, `numerador` e `denominador`, do tipo `int`, e faça o seguinte:
 - a. Escreva uma função `setFraction` que recebe dois argumentos do tipo `int` (o numerador e o denominador), e retorna os valores em um tipo `Fraction`.
 - b. Escreva uma função `multFraction` que recebe dois argumentos do tipo `Fraction`, multiplica, armazena o resultado em uma outra variável do tipo `Fraction` e retorna esta variável. Lembrando: $\frac{4}{5} \times \frac{3}{2} = \frac{(4 \times 3)}{(5 \times 2)}$

Avançado

- Variáveis do tipo estrutura podem ser declaradas como arranjos (vetores) e assim, tem-se uma lista
 - `struct tdata Data[10];`
- O acesso aos elementos é da mesma forma usada para variáveis escalares:
 - `Data[0].dia=06;`
 - `Data[0].mes=12;`
 - `Data[0].ano=2021;`

Avançado

- Crie uma estrutura representando os alunos de um determinado curso. A estrutura deve conter a **matrícula** do aluno, **nome**, **nota da primeira prova**, **nota da segunda prova** e **nota da terceira prova**.
 - Permita ao usuário entrar com os dados de 5 alunos.
 - Encontre o aluno com maior nota da primeira prova.
 - Encontre o aluno com maior média geral.
 - Encontre o aluno com menor média geral.
 - Para cada aluno diga se ele foi aprovado ou reprovado, considerando o valor 6 para aprovação.

TAD

Tipo Abstrato de Dado

Introdução

- Se considerarmos a definição de estruturas complexas da última aula (**structs**), podemos criar tipos compostos e que representam de maneira mais fidedigna elementos do mundo real
 - Lembrando que a linguagem C tem apenas o conjunto restrito de tipos: int, float, char ... e operações sobre eles: +, -, * ...
- Utilizando **structs**, podemos criar tipos mais complexos e operações que possam ser executadas sobre estes tipos
 - Podemos criar um **tipo fração** (numerador e denominador) e **operações** sobre o tipo criado (ou o tipo abstrato de dado criado)

```
tipofrac myfrac1, myfrac2, myfrac3;  
myfrac1=atrib_fracao(4,8);  
myfrac2=atrib_fracao(3,8);  
myfrac3=soma_frac(myfrac1,myfrac2);  
imp_frac(myfrac3); // pode imprimir 7/8
```

```
tipofrac atrib_fracao (int n, int d)  
{  
    tipofrac f;  
    f.numerador=n;  
    f.denominador=d;  
    return f;  
}
```

Introdução

- Imagine a representação de ponto em um plano de duas dimensões
 - Teremos duas coordenadas (x,y)

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```


Introdução

- Imagine a representação de ponto em um plano de duas dimensões
 - Teremos duas coordenadas (x,y)
 - Como calcular a distância euclidiana? (distância entre os pontos)

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

Introdução

- Imagine a representação de ponto em um plano de duas dimensões
 - Teremos duas coordenadas (x,y)
 - Como calcular a distancia euclidiana? $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

```
int main() {  
    Ponto p1 = {-2, 3}, p2 = {-5, -9};  
    double distancia;  
  
    distancia = pow((p2.x - p1.x), 2) + pow((p2.y - p1.y), 2);  
    distancia = sqrt(distancia);  
  
    printf("%.5lf \n", distancia);  
  
    return 0;  
}
```

Introdução

- Imagine a representação de ponto em um plano de duas dimensões
 - Teremos duas coordenadas (x,y)
 - Como calcular a distancia euclidiana?

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

```
double distanciaEuclidiana( Ponto p1, Ponto p2){  
    double distancia;  
    distancia = pow((p2.x - p1.x),2) + pow((p2.y - p1.y),2);  
    distancia = sqrt(distancia);  
    return distancia;  
}
```

Introdução

- Imagine a representação de ponto em um plano de duas dimensões
 - Teremos duas coordenadas (x,y)
 - Como calcular a distancia euclidiana?

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

```
double distanciaEuclidiana( Ponto p1, Ponto p2){  
    double distancia;  
    distancia = pow((p2.x - p1.x),2) + pow((p2.y - p1.y),2);  
    distancia = sqrt(distancia);  
    return distancia;  
}
```

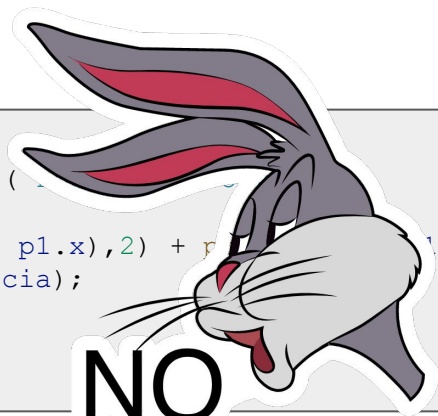
Se eu precisar utilizar essa função de novo, eu tenho que copiar colar no novo programa?

Introdução

- Imagine a representação de ponto em um plano de duas dimensões
 - Teremos duas coordenadas (x,y)
 - Como calcular a distancia euclidiana?

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

```
double distanciaEuclidiana(  
    double distancia;  
    distancia = pow((p2.x - p1.x),2) + pow((p2.y - p1.y),2);  
    distancia = sqrt(distancia);  
    return distancia;  
}
```



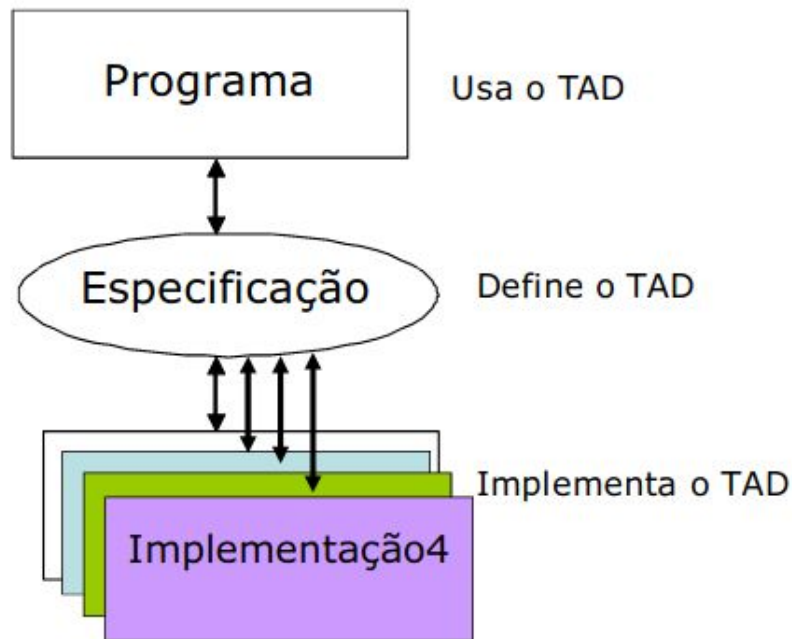
Se eu precisar utilizar essa função de novo, eu tenho que copiar e colar no novo programa?

TAD

- Utilizamos nesses casos os Tipo Abstrato de Dado (TAD)
 - TAD especifica o tipo de dado (domínio de operações) sem referência a detalhes da implementação
 - Permite maior flexibilidade no desenvolvimento, principalmente na manutenção do código
 - O programador não sabe como o TAD foi implementado, as implementações ficam “escondidas”
- O TAD especifica tudo o que precisa saber para usar um determinado tipo
- O TAD divide o sistema em:
 - Programas de Usuário
 - Implementação

TAD

- Utilizamos nesse contexto:
 - TAD específico para cada implementação
 - Permite maior reutilização de código
 - O programador não precisa lidar com detalhes “escondidos”
- O TAD especifica a interface a ser utilizada
- O TAD divide o trabalho em:
 - Programas de aplicação
 - Implementações



TAD)

ência a detalhes da

a manutenção do código
implementações ficam

n determinado tipo

TAD

- O TAD é representado por dois documentos (programas)
 - Especificação
 - Chamado de arquivo de cabeçalho em C ([header](#))
 - É nomeado como [nome_arquivo.h](#)
 - Implementação
 - Efetivamente implementa as funções declaradas no cabeçalho
 - Geralmente tem o mesmo nome do .h mas .c ([nome_arquivo.c](#))

TAD

- Cabeçalho

planoCartesiano.h

```
typedef struct {  
    int x;  
    int y;  
} Ponto;  
  
double distanciaEuclidiana( Ponto p1, Ponto p2);  
void setPonto( Ponto *p1, int x, int y);
```

TAD

- implementação

planoCartesiano.c

```
#include <stdlib.h>
#include <math.h>
#include "planoCartesiano.h"

double distanciaEuclidiana( Ponto p1, Ponto p2){
    double distancia;
    distancia = pow((p2.x - p1.x),2) + pow((p2.y - p1.y),2);
    distancia = sqrt(distancia);
    return distancia;
}

void setPonto( Ponto *p1, int x, int y){
    p1->x=x;
    p1->y=y;
}
```

TAD

- Uso por outros programas

```
#include <stdlib.h>
#include <math.h>
#include "planoCartesiano.h"
int main()
{
    Ponto pt1, pt2;
    setPonto(&pt1,10,15);
    setPonto(&pt2,60,35);
    printf("%.5lf",distanciaEuclidiana(pt1,pt2));
    return 0;
}
```

- Compilação `gcc -Wall principal.c planoCartesiano.c -lm -o principal` (Linux)
- Compilação `gcc -Wall principal.c planoCartesiano.c -o principal` (Windows)

TAD

- Exercício
 - Implemente um TAD que represente frações e as operações sobre as mesmas
 - Atribuição
 - Multiplicação
 - Divisão
 - Opcional:
 - Adição e subtração (tem que calcular o MMC)

```
int mmc(int a,int b)
{
    int div;
    if(b == 0) return a;
    else
        div = (a*b)/(mdc(a,b));
    return (div);
}
```