

Estrutura de Dados

1º semestre de 2025

Tipos abstratos de dados

Tema #01

Professor Marcelo Eustáquio



Estrutura de dados

Estruturas de dados e algoritmos estão intimamente ligados...

Não se pode estudar estrutura de dados sem considerar os algoritmos associados a elas, assim como a escolha dos algoritmos, em geral, depende da representação da estrutura de dados.



Para resolver um problema é necessário escolher uma abstração da realidade, em geral mediante a definição de um conjunto de dados que representa uma situação real. A seguir, deve ser escolhida a forma de representar esses dados.

Estrutura de dados

A escolha da representação dos dados é determinada, entre outras, pelas operações a serem realizadas sobre os dados. Para ilustrar, considere a operação de adição:

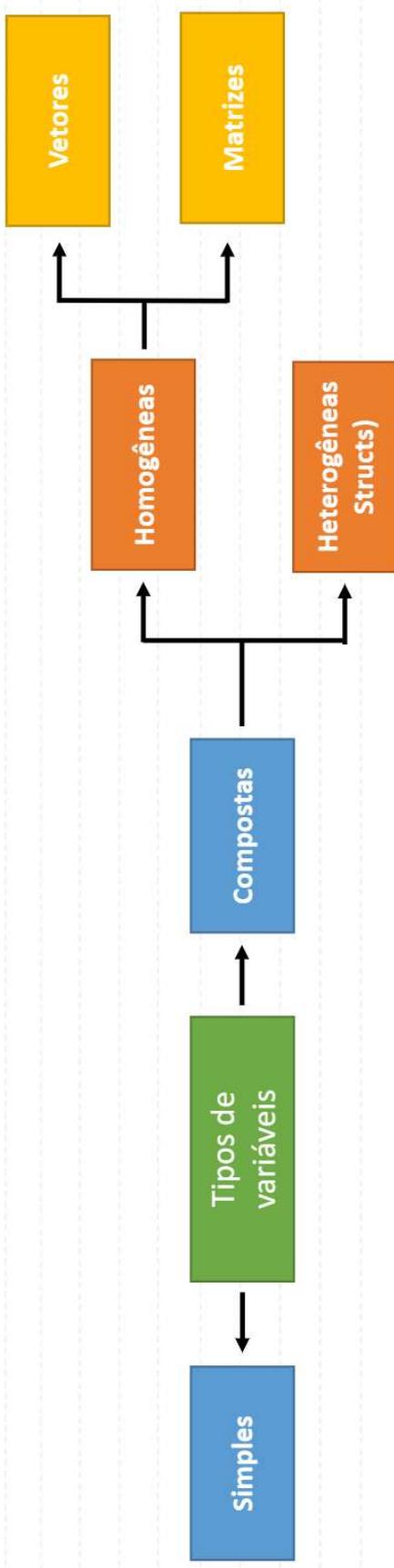
- 1** Para pequenos números, uma boa representação é por meio de barras verticais (caso em que a operação de adição é bastante simples).
- 2** Já a representação por dígitos decimais requer regras relativamente complicadas, as quais devem ser memorizadas.
- 3** Entretanto, quando consideramos a adição de grandes números é mais fácil a representação por dígitos decimais (devido ao princípio baseado no peso relativo da posição de cada dígito).

Exemplo:

Qual seria a melhor abordagem para determinar o valor de $8 + 7$? E para calcular o resultado da soma $182.352 + 176.439$?

Estrutura de dados

Existem dois tipos principais de dados: os tipos simples e os tipos estruturados que, em geram definem uma coleção de valores simples ou um agregado de valores de tipos diferentes.



E então, o que é TAD?

Uma **tipo abstrato de dados** (TAD) é uma forma de definir **novo tipo** de dados juntamente com as **operações** que manipulam esse novo tipo de dado, correspondendo a uma estruturação conceitual de dados e refletindo o relacionamento lógico entre eles.

Tipo Abstrato de
Dados

Conjunto de tipos
de dados

Operações a serem
realizadas

Ou seja:

- TAD é um modelo matemático, acompanhado das operações definidas sobre o modelo.
- Um exemplo é o conjunto \mathbb{Z} acompanhado das operações de adição, subtração, multiplicação e divisão.
- TAD's são usados como base para projetos de algoritmos.
- Implementação de algoritmos envolve representação do TAD em termos de tipos de dados e operações.



A construção de TAD's é importante na programação orientada por objetos, dado que a representação do modelo matemático do TAD é realizada mediante uma estrutura de dados (**struct**).

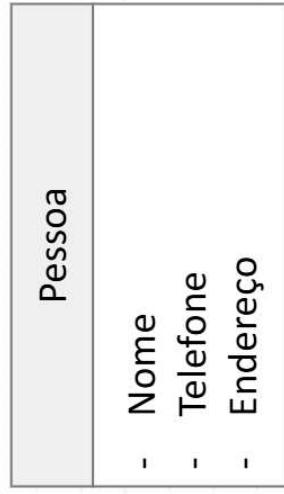


O TAD encapsula tipos de dados, e a definição do tipo e das operações ficam localizadas numa seção específica do código-fonte (e do programa).

Implementação do TAD

Na linguagem C, utilizamos o conceito de estrutura para definir tipos abstratos de dados.

- Uma estrutura agrupa várias variáveis numa só.
- Funciona como uma ficha pessoal que tenha nome, telefone e endereço.
- A ficha seria uma estrutura, que serve para agrupar um conjunto de dados não similares.



Representação da classe Pessoa, muito importante nos diagramas de classe da UML.

Implementando o TAD

Fração



Frações...

Para implementar o tipo abstrato de dados fração, demos primeiro responder a seguinte pergunta: o que define uma fração, ou seja, o que torna uma fração diferente da outra. E a resposta é imediata: cada fração é definida a partir do **numerador** e do **denominador**.

Fração
- numerador
- denominador

Logo:

- No TAD fração devem constar, como atributos, os inteiros Numerador e Denominador.
- A instrução typedef pode ser empregada para simplificar a notação.

Frações...

Para implementar o tipo abstrato de dados fração, demos primeiro responder a seguinte pergunta: o que define uma fração, ou seja, o que torna uma fração diferente da outra. E a resposta é imediata: cada fração é definida a partir do **numerador** e do **denominador**.

```
typedef struct {
    int numerador;
    int denominador;
} Fracao;
```

Atenção:

- 1** Foi definido um tipo de dado personalizado chamado Fracao usando typedef e struct.
- 2** Evita a necessidade de escrever struct Fracao toda vez, tornando o código mais limpo.
- 3** Facilita a reutilização do tipo em funções e estruturas mais complexas.

Frações...

Agora, para continuar, devem ser estabelecidas as **operações** (**funções**) que manipularão frações. Uma possibilidade é mostrada a seguir, que incluem a inicialização dos atributos, além de operações aritméticas básicas e de operações que manipulam os atributos numerador e denominador.

Criar fração	Somar frações	Subtrair frações	Multiplicar frações	Dividir frações
Simplificar fração	Converter fração	Exibir fração	Comparar fração	Calcular potência

Essas são apenas algumas operações (e outras podem ser estabelecidas). A seguir, vamos implementar algumas delas, a começar por “Criar fração”.

Criar fração

O objetivo da operação “*Criar fração*” é inicializar os atributos de uma variável do tipo *Fração*, com valores passados como parâmetro.

```
Fracao criarFracao(int numerador, int denominador) {  
    Fracao f;  
    f.numerador = numerador;  
    f.denominador = denominador;  
    return f;  
}
```

Observações:

- Esta função desempenha o papel de “construtor”, em analogia à orientação por objetos.
- Na inicialização, é importante que sejam verificadas as restrições de domínio (*denominador* ≠ 0).

```
Fracao criarFracao(int numerador, int denominador) {  
    Fracao f;  
    f.numerador = numerador;  
    f.denominador = denominador;  
    return f;  
}
```



Como utilizar a função criarFracao para criar a fração
 $3/7$ e atribuí-la à variável X?

$$X = \frac{3}{7}$$



```
Fracao X = criarFracao(3, 7);
```

Simplificar fração

As frações podem ser classificadas em redutíveis (que podem ser simplificadas) ou irredutíveis (que não podem ser simplificadas). Por exemplo, a fração **12/20** pode ser simplificada e escrita como **3/5** a partir da divisão do numerador e do denominador por **4**.

$$\frac{12}{20} = \frac{12 \div 4}{20 \div 4} = \frac{3}{5}$$

No caso, 4 é o **máximo divisor comum** entre **numerador** e **denominador** da fração, e uma possibilidade, ineficiente, de calcular o MDC entre numerador e denominador é apresentada a seguir:

```
int aux = (F.numerador > F.denominador) ? F.denominador : F.numerador;

for (int i = 1; i <= aux; i++)
    if (F.numerador % i == 0 && F.denominador % i == 0)
        MDC = i;
```

Simplificar fração

Eis a função “*SimplificarFracao*” completa...

```
Fracao simplificarFracao(Fracao F) {  
  
    int MDC = 1;  
    int aux = (F.numerador > F.denominador) ? F.denominador : F.numerador;  
  
    for (int i = 1; i <= aux; i++)  
        if (F.numerador % i == 0 && F.denominador % i == 0)  
            MDC = i;  
  
    F.numerador = F.numerador / MDC;  
    F.denominador = F.denominador / MDC;  
  
    return F;  
}
```

Simplificar fração



Usando as funções implementadas, mostre como declarar uma variável do tipo **"Fracao"**, como inicializa-la com a 48 / 72 e, por fim, mostre como simplifica-la.

Observação:

O maior divisor comum entre 48 e 72 é igual a 24 e, após simplificar 48/72, obtém-se 2/3.

$$\frac{48}{72} = \frac{48 \div 24}{72 \div 24} = \frac{2}{3}$$

Somar frações

Normalmente, a soma de frações se dá por meio da redução dessas frações a um denominador comum (o que pode ser feito a partir do cálculo do MMC dos denominadores dessas frações). No entanto, uma possibilidade é desenvolver o cálculo conforme exemplificado a seguir com a soma de **5/6 e 4/9**:

$$\frac{5}{6} + \frac{4}{9} = \frac{5 \cdot 9}{6 \cdot 9} + \frac{4 \cdot 6}{6 \cdot 9} = \frac{45}{54} + \frac{24}{54} = \frac{69}{54} = \frac{69 \div 3}{54 \div 3} = \frac{23}{18}$$



- O denominador da fração soma é o produto dos denominadores das frações-parcela.
- Os numeradores são obtidos a partir de divisões e multiplicações feitas com o denominador comum.
- Por fim, com denominadores comuns, é determinada a soma dos numeradores.

Somar frações

Por fim, eis a função “**SomarFracoes**” completa...

```
// Função para somar duas frações

Fracao SomarFracao(Fracao F, Fracao G) {
    Fracao resultado;

    resultado.numerador = (F.numerador * G.denominador) + (G.numerador * F.denominador);
    resultado.denominador = F.denominador * G.denominador;
    resultado = simplificarFracao(resultado);

    return resultado;
}
```

Somar frações



Utilize a função “*SomarFracoes*” para determinar o valor de $\frac{3}{8} + \frac{5}{6} + \frac{7}{12}$.

Subtrair frações

Assim como a adição, a subtração de frações também se dá por meio da redução dessas frações a um denominador comum, que será determinado a partir do produto dos denominadores das frações que compõem a subtração.

$$\frac{5}{6} - \frac{4}{9} = \frac{5 \cdot 9}{6 \cdot 9} - \frac{4 \cdot 6}{6 \cdot 9} = \frac{45}{54} - \frac{24}{54} = \frac{21}{54} = \frac{21 \div 3}{54 \div 3} = \frac{7}{18}$$



- O denominador da fração subtração é o produto dos denominadores das frações-diferença.
- Os numeradores são obtidos a partir de divisões e multiplicações feitas com o denominador comum.
- Por fim, com denominadores comuns, é determinada a subtração dos numeradores.

Subtrair frações

Por fim, eis a função “**SubtrairFracoes**” completa...

```
// Função para subtrair duas frações

Fracao subtrairFracao(Fracao F, Fracao G) {
    Fracao resultado;

    resultado.numerador = (F.numerador * G.denominador) - (G.numerador * F.denominador);
    resultado.denominador = F.denominador * G.denominador;
    resultado = simplificarFracao(resultado);

    return resultado;
}
```



Subtrair frações

Utilize as funções “*SomarFracoes*” e “*SubtrairFracoes*” para determinar o valor de $\frac{3}{4} - \frac{7}{9} + \frac{5}{12}$.

Multiplicar frações

Na multiplicação de frações, o resultado é uma fração em que o numerador é o produto dos numeradores e o denominador é o produto dos denominadores. A seguir, será realizada a **multiplicação** das frações **5/7** e **21/20**:

$$\frac{5}{7} \times \frac{21}{20} = \frac{5 \cdot 21}{7 \cdot 20} = \frac{105}{140} = \frac{105 \div 35}{140 \div 35} = \frac{3}{4}$$

Multiplicar frações

Por fim, eis a função “**MultiplicarFracoes**” completa...

```
// Função para multiplicar duas frações

Fracao multiplicarFracao(Fracao F, Fracao G) {

    Fracao resultado;

    resultado.numerador = F.numerador * G.numerador;
    resultado.denominador = F.denominador * G.denominador;
    resultado = simplificarFracao(resultado);

    return resultado;
}
```



Multiplicar frações

Utilize as funções implementadas até o momento para determinar o valor de $\frac{3}{4} \times \frac{7}{9} + \frac{5}{12}$.

Dividir frações

Na divisão entre frações, o resultado é obtido multiplicando a primeira fração pelo inverso da segunda fração. A seguir, será realizada a **divisão** entre as frações **5/7** e **21/20**:

$$\frac{5}{7} \div \frac{21}{20} = \frac{5}{7} \times \frac{20}{21} = \frac{5 \cdot 20}{7 \cdot 21} = \frac{100}{147}$$

Observações:

- Por se tratar de divisão, é importante que
- Na inicialização, é importante que sejam verificadas as restrições de domínio (denominador $\neq 0$).

Dividir frações

Por fim, eis a função “*dividirFracoes*” completa...

```
// Função para dividir duas frações

Fracao dividirFracao(Fracao F, Fracao G) {
    Fracao resultado;

    resultado.numerador = F.numerador * G.denominador;
    resultado.denominador = F.denominador * G.numerador;
    resultado = simplificarFracao(resultado);

    return resultado;
}
```



Dividir frações

Utilize as funções implementadas até o momento para determinar o valor de $\frac{3}{4} \div \frac{7}{9} - \frac{5}{12}$.

Estruturando o código-fonte.



Arquivo Único.c

ou

main.c

+

fracção.c

+

fracção.h

Para compilar, no prompt de comando, deve-se digitar **gcc fraccao.c main.c -o fraccao.exe**. A seguir, para executar, digite **./fraccao.exe** ou **./fraccao**

fracao.h

```
#ifndef FRACAO_H  
#define FRACAO_H
```

Essa diretiva significa "**if not defined**" (se não definido) e é usada para evitar a inclusão repetida de um arquivo de cabeçalho em um programa.

```
typedef struct {  
    int numerador;  
    int denominador;  
} Fracao;
```

Funcionamento:

1) Ao escrever **#ifndef FRACAO_H**, é verificado se o símbolo **FRACAO_H** não está definido antes.

2) Se o **FRACAO_H** não estiver definido, o pré-processador C define esse símbolo usando a diretiva **#define**. **FRACAO_H** e o código entre **#ifndef** e **#endif** serão incluído no programa.

3) Se **FRACAO_H** já estiver definido (ou seja, o arquivo de cabeçalho já foi incluído), o código entre **#ifndef** e **#endif** será ignorado, evitando a inclusão repetida.

fracao.c

Nesse arquivo estarão as implementações das funções cujo escopo (assinatura) foram definidos no arquivo de cabeçalho fracao.h.

```
#include <stdio.h>
#include "fracao.h"
```

```
Fracao criarFracao(int numerador, int denominador) {
    Fracao f;
    f.numerador = numerador;
    f.denominador = denominador;
    return f;
}
```

// Função para somar duas frações

```
Fracao somarFracao(Fracao a, Fracao b) {  
    Fracao resultado;  
    resultado.numerador = (a.numerador * b.denominador) + (b.numerador * a.denominador);  
    resultado.denominador = a.denominador * b.denominador;  
    return resultado;  
}
```

// Função para subtrair duas frações

```
Fracao subtrairFracao(Fracao a, Fracao b) {  
    Fracao resultado;  
    resultado.numerador = (a.numerador * b.denominador) - (b.numerador * a.denominador);  
    resultado.denominador = a.denominador * b.denominador;  
    return resultado;  
}
```

```
// Função para multiplicar duas frações

Fracao multiplicarFracao(Fracao a, Fracao b) {
    Fracao resultado;
    resultado.numerador = a.numerador * b.numerador;
    resultado.denominador = a.denominador * b.denominador;
    return resultado;
}

// Função para dividir duas frações

Fracao dividirFracao(Fracao a, Fracao b) {
    Fracao resultado;
    resultado.numerador = a.numerador * b.denominador;
    resultado.denominador = a.denominador * b.numerador;
    return resultado;
}
```

```
// Função para simplificar uma fração
```

```
Fracao simplificarFracao(Fracao f) {
    int gcd = mdc(f.numerador, f.denominador);
    f.numerador /= gcd;
    f.denominador /= gcd;
    return f;
}
```

```
// Função para calcular o máximo divisor comum (MDG) de dois números
```

```
int mdc(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

main.c

Arquivo contendo a função principal (**main**).

```
#include <stdio.h>
#include <stdlib.h>

#include "fracao.h"
```

```

int main() {
    Fracao f1 = criarFracao(1, 2);
    Fracao f2 = criarFracao(3, 4);

    Fracao soma = somarFracao(f1, f2);
    Fracao subtracao = subtrairFracao(f1, f2);
    Fracao multiplicacao = multiplicarFracao(f1, f2);
    Fracao divisao = dividirFracao(f1, f2);

    soma = simplificarFracao(soma);
    subtracao = simplificarFracao(subtracao);
    multiplicacao = simplificarFracao(multiplicacao);
    divisao = simplificarFracao(divisao);

    printf("Soma: %d/%d\n", soma.numerador, soma.denominador);
    printf("Subtração: %d/%d\n", subtracao.numerador, subtracao.denominador);
    printf("Multiplicação: %d/%d\n", multiplicacao.numerador, multiplicacao.denominador);
    printf("Divisão: %d/%d\n", divisao.numerador, divisao.denominador);
    system("pause");
    return 0;
}

```

ArquivoÚnico.c

ou

main.c

+

fração.c

+

fração.h

Para compilar, no prompt de comando, deve-se digitar **gcc fracao.c main.c -o fracao.exe**. A seguir, para executar, digite **./fracao.exe** ou **./fracao**

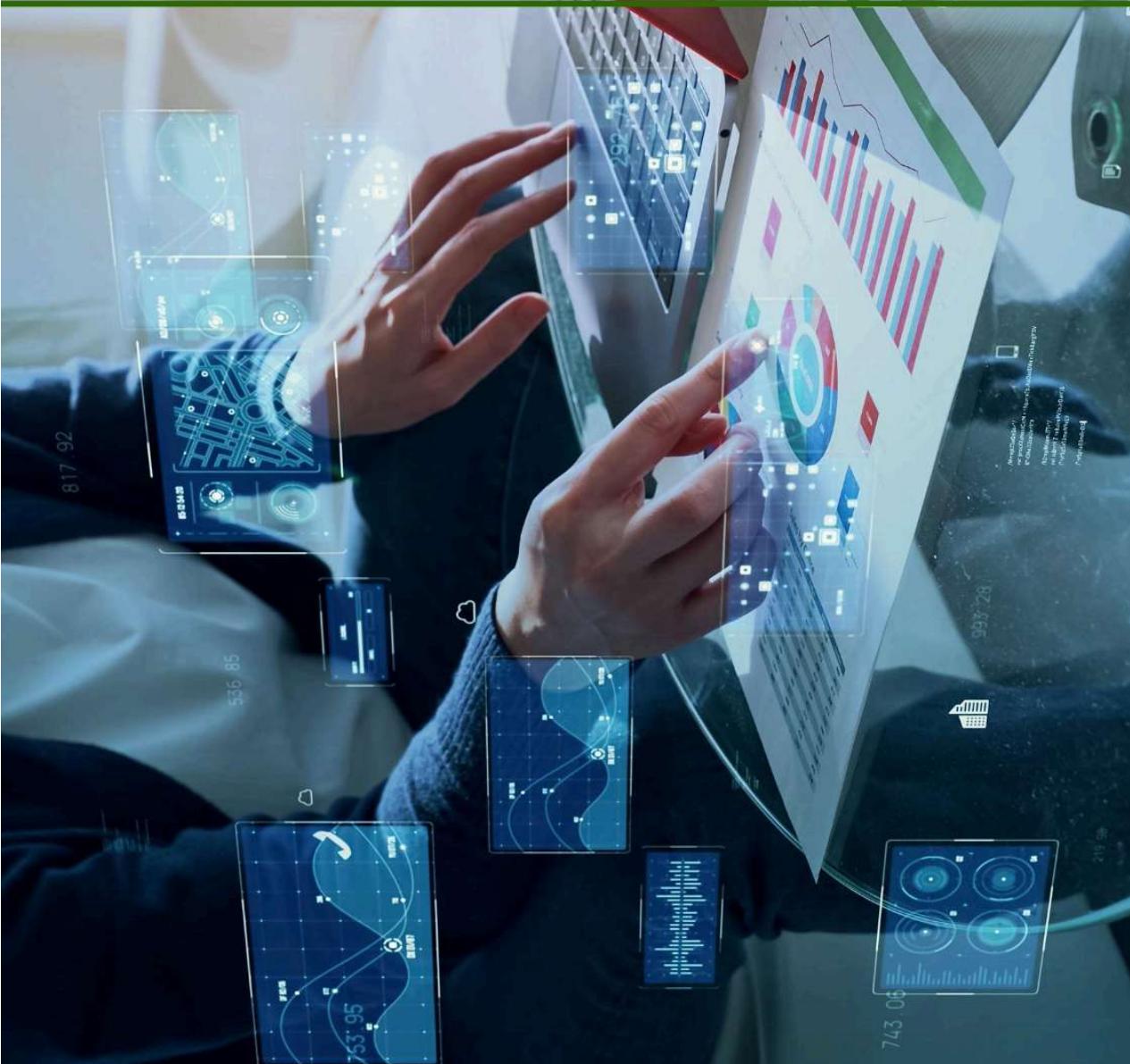
Estrutura de Dados

1º semestre de 2025

Tipos abstratos de dados

Exercícios

Professor Marcelo Eustáquio



Q01

Em um banco de dados, são armazenadas os seguintes dados:

dados.csv

ID	Nome	Idade
1	Ana	23
2	João	35
3	Maria	28

Agora, responda:

- 1) Nesse caso, como você definiria o tipo abstrato de dados Pessoa?
- 2) Cite pelo menos três operações que manipulariam os elementos do TAD Pessoa.
- 3) Codifique/implemente uma das operações propostas na pergunta 2.

O tipo abstrato de dados Pessoa poderia ser implementado da seguinte forma (a structs ListaPessoas é uma construção para agrupar todas as pessoas cadastradas, além de tamanho e capacidade do conjunto, ou seja, não impacta na definição de cada pessoa propriamente dita).

```
typedef struct {
    int id;
    char nome[NOME_MAX];
    int idade;
} Pessoa;

typedef struct {
    Pessoa *dados; // Usando alocação dinâmica, dados pode ser interpretado como vetor de Pessoas
    int tamanho;
    int capacidade;
} ListaPessoas;
```

A seguir, são indicadas funções (operações) que manipulam esses dados.

```
void initLista(ListaPessoas *L);
void destruirLista(ListaPessoas *L);
void garantirCapacidade(ListaPessoas *L);
int inserirPessoa(ListaPessoas *L, Pessoa p);
Pessoa* buscarPessoaPorId(ListaPessoas *L, int id);
int atualizarIdade(ListaPessoas *L, int id, int novaIdade);
int removerPessoaPorId(ListaPessoas *L, int id);
void imprimirLista(const ListaPessoas *L);
```

Uma implementação para imprimirLista seria:

```
void imprimirLista(const ListaPessoas *L) {
    size_t i;
    printf("ID, Nome, Idade\n");
    for (i = 0; i < L->tamanho; ++i)
        printf("%d,%s,%d\n", L->dados[i].id, L->dados[i].nome, L->dados[i].idade);
}
```

Q02

A seguir, é mostrado como um sistema de gestão de estoque de uma loja armazena informações:

vendas.csv

Produto	Quantidade	Preço
Camisa	10	30.50
Calça	5	80.00
Tênis	8	120.00

Agora, faça o que se pede:

- 1) Defina o TAD Item.
- 2) Cite pelo menos três operações que manipulariam os elementos do TAD Item.
- 3) Como você trataria uma compra de 1 ou mais itens nessa loja?

O tipo abstrato de dados Item poderia ser implementado da seguinte forma:

```
typedef struct {
    char produto[NOME_MAX];
    int quantidade;
    double preco;
} Item;

typedef struct {
    Item *dados;
    int tamanho;
    int capacidade;
} Estoque;

typedef struct {
    char produto[NOME_MAX];
    int quantidade;
} CarrinhoItem;
```

A seguir, são indicadas funções (operações) que manipulam esses dados.

```
void initEstoque(Estoque *E);
void destruirEstoque(Estoque *E);
void garantirCapacidade(Estoque *E);
int buscarIndexPorNome(Estoque *E, const char *nome);
int inserirItem(Estoque *E, Item it);
int atualizarPreco(Estoque *E, const char *produto, double novoPreco);
int reporEstoque(Estoque *E, const char *produto, int qtd);
int venderUm(Estoque *E, const char *produto, int qtd);
void imprimirEstoque(const Estoque *E);
int comprarItens(Estoque *E, const CarrinhoItem *cart, int n, double *total_out);
```

Na compra de 1 ou mais itens, a função **comprarItens(Estoque*, CarrinhoItem[], n, &total)** (a) verifica TODOS os itens (existência e saldo) antes; (b) se algum falhar, NÃO altera o estoque (transação abortada); e (c) se tudo ok, debita as quantidades e retorna o total (double).

Q03

Em uma universidade, as notas dos alunos estão armazenadas em um banco de dados relacional da seguinte forma:

alunos.csv

```
Nome,Nota1,Nota2  
Carlos,6.5,8.0  
Fernanda,7.0,7.5  
Lucas,5.0,6.0
```

Agora, faça o que se pede:

Implemente o TAD Aluno, indicando dados e operações que julgar importantes.

Definição do tipo abstrato de dados **Aluno**. Para armazenamento das informações, foi definida a structs **Aluno**, além da structs **Turma**, que contém o ponteiro (ou vetor de Alunos) **dados**.

```
typedef struct {
    char nome[NOME_MAX];
    double nota1;
    double nota2;
} Aluno;

typedef struct {
    Aluno *dados;
    int tamanho;
    int capacidade;
} Turma;
```

Para refletir: quantos bytes ocupa, em memória, uma variável do tipo **Aluno**? Para descobrir a resposta, use `sizeof(Aluno)`.

Algumas **operações** que poderiam compor o tipo abstrato de dados **Aluno** tem protótipos definidos a seguir (a lista aqui apresentada é **exemplificativa**).

```
void initTurma(Turma *T);
void destruirTurma(Turma *T);
void garantirCapacidade(Turma *T);
int buscarIndexPorNome(Turma *T, char *nome);
int inserirAluno(Turma *T, Aluno a);
int atualizarNotas(Turma *T, char *nome, double n1, double n2);
double mediaAluno(Aluno *a);
int statusAluno(Aluno *a, double corte);
double mediaTurma(Turma *T);
Aluno* alunoMaiorMedia(Turma *T);
void ordenarPorMediaDesc(Turma *T);
void imprimirBoletim(Turma *T, double corte);
```

Para refletir: que outras funções (operações) poderiam ser definidas?

Q04

O departamento financeiro de uma empresa guarda informações relativas à folha de pagamentos dos funcionários em um banco de dados relacional, conforme é mostrado a seguir:

funcionarios.csv

```
Nome,Idade,Salário
Roberto,30,2500.00
Juliana,45,3000.50
```

Agora, faça o que se pede:

Implemente o TAD Funcionário e mostre como você implementaria a operação calcularFGTS relativo a um determinado mês. Como você faria para testar essa operação?

Definição do tipo abstrato de dados **Funcionário**. Para armazenamento das informações, foi definida a structs Aluno, além da structs Turma, que contém o ponteiro (ou vetor de Funcionários) **dados**.

```
typedef struct {
    char nome[NOME_MAX];
    int idade;
    double salario; /* salário base mensal */
} Funcionario;

typedef struct {
    Funcionario *dados;
    int tamanho;
    int capacidade;
} Folha;
```

Algumas **operações** que poderiam compor o tipo abstrato de dados **Funcionário** tem protótipos definidos a seguir (a lista aqui apresentada é **exemplificativa**).

```
void initFolha(Folha *F) ;
void destruirFolha(Folha *F) ;
void garantirCapacidade(Folha *F) ;
int buscarIndexPorNome(Folha *F, char *nome) ;
int inserirFuncionario(Folha *F, Funcionario x) ;
int atualizarSalario(Folha *F, char *nome, double novoSal) ;
void imprimirFuncionarios(Folha *F) ;
double calcularFGTS(const Funcionario *f, double adicionaisMes, double aliquota) ;
double calcularFGTS8(const Funcionario *f, double adicionaisMes) ;
```

Uma estratégia (mais generalista) para cálculo do FGTS é apresentada a seguir: aqui, optou-se pela passagem da alíquota como parâmetro para possibilitar reuso. Para utilizá-la uma possibilidade é **calcularFGTS(f, adicionaisMes, 0.08)**

```
/* calcularFGTS: remuneração do mês = salário base + adicionais (horas extra, etc.) */

double calcularFGTS(const Funcionario *f, double adicionaisMes, double aliquota) {
    double remuneracao = f->salario + adicionaisMes;
    if (remuneracao < 0) remuneracao = 0; /* proteção básica */
    return remuneracao * aliquota;
}
```

*Observação: A alíquota “padrão” do FGTS (**8%** da remuneração mensal) está no art. 15, caput, da Lei nº **8.036/1990** – a lei do FGTS. O próprio governo resume assim: o empregador deve depositar 8% da remuneração do mês anterior na conta do trabalhador (com 2% para contratos de aprendizagem, conforme §7º do mesmo artigo). Para empregados domésticos, no eSocial, continuam sendo 8% de FGTS + 3,2% de antecipação da multa rescisória.*

Q05

Uma livraria guarda informações dos livros disponíveis para venda em um banco de dados relacional, conforme é exemplificado a seguir:

livros.csv

```
LivroID, Titulo, Autor, Genero  
1, Dom Quixote, Miguel de Cervantes, "Romance, Aventura"  
2, O Senhor dos Anéis, J.R.R. Tolkien, "Fantasia, Aventura, Épico"
```

Agora, faça o que se pede:

- 1) Defina os dados do TAD Livro.
- 2) Qual a diferença entre atributos monovalorados e atributos multivalorados?
- 3) Como você armazenaria o item 2 em uma variável do TAD Livro que fora implementado?
- 4) Como verificaria se o Gênero do item 1 contém Aventura?

Definindo os dados do TAD Livro.

```
#define TXT_MAX 128
#define MAX_GEN 10 /* no máx. 10 gêneros por livro */
#define GEN_MAX 24 /* tamanho máximo de cada nome de gênero */

typedef struct {
    int id; /* LivroID (monovalorado) */
    char titulo[TXT_MAX]; /* Título (monovalorado) */
    char autor[TXT_MAX]; /* Autor (monovalorado) */
    char generos[MAX_GEN][GEN_MAX]; /* Gênero(s) (multivalorado) */
    int nGeneros; /* quantos gêneros foram setados */
} Livro;
```

Observação: TXT_MAX, MAX_GEN e GEN_MAX são constantes e não podem ser alteradas durante a execução (em fase de pré-compilação, são substituídas pelos valores indicados).

A diferença entre atributos **monovalorados** e atributos **multivvalorados** ...

Observação:

Monovvalorado: assume um único valor por entidade no modelo (ex.: LivroID, Titulo, Autor).

Multivvalorado: pode assumir vários valores simultâneos (ex.: Genero: “Fantasia”, “Aventura”, “Épico”).

No TAD acima, **generos** é um vetor de strings com contador nGeneros, representando um atributo multivvalorado.

Como você armazenaria o item 2 em uma variável do TAD Livro que fora implementado?

```
int main(void) {  
    Livro l12;  
    initLivro(&l12, 2, "O Senhor dos Aneis", "J.R.R. Tolkien");  
    addGenero(&l12, "Fantasia");  
    addGenero(&l12, "Aventura");  
    addGenero(&l12, "Epic"); /* sem acento por compatibilidade */  
  
    /* impressão rápida para conferência */  
  
    printf("ID=%d, Titulo=%s, Autor=%s, Generos:", l12.id, l12.titulo, l12.autor);  
    for (int i = 0; i < l12.nGeneros; ++i) printf(" %s%s", l12.generos[i], (i+1<l12.nGeneros?" ":""));  
    printf("\n");  
}
```

Como verificaria se o Gênero do item 1 contém Aventura?

```
int main(void) {  
  
    Livro l1;  
    initLivre(&l1, 1, "Dom Quixote", "Miguel de Cervantes");  
    addGenero(&l1, "Romance");  
    addGenero(&l1, "Aventura");  
  
    if (hasGenero(&l1, "Aventura")) printf("Item 1 contém o gênero 'Aventura' .\n");  
    else printf("Item 1 NAO contém 'Aventura' .\n");  
    return 0;  
}
```

As funções adicionais (referenciadas acima) estão detalhadas a seguir:

```

void initLivro(Livro *L, int id, const char *titulo, const char *autor) {
    L->id = id;
    strcpy(L->titulo, titulo, TXT_MAX-1); L->titulo[TXT_MAX-1] = '\0';
    strcpy(L->autor, autor, TXT_MAX-1); L->autor[TXT_MAX-1] = '\0';
    L->nGeneros = 0;
}

int addGenero(Livro *L, const char *g) {
    if (L->nGeneros >= MAX_GEN) return 0;
    strcpy(L->generos[L->nGeneros], g, GEN_MAX-1);
    L->generos[L->nGeneros][GEN_MAX-1] = '\0';
    L->nGeneros++;
    return 1;
}

int hasGenero(const Livro *L, const char *g) {
    int i;
    for (i = 0; i < L->nGeneros; +i) if (strcmp(L->generos[i], g) == 0) return 1;
    return 0;
}

```