

Notas de aula – AED 1 – TAD (tipo abstrato de dados)  
Prof. Jefferson T. Oliva

Um programa em C pode ser dividido em vários arquivos fontes. Como vocês devem ter visto, é possível criar cabeçalhos (arquivos .h) para organizar o seu código em módulos e bibliotecas. Para cada arquivo .h deve haver um outro .c, com o mesmo nome e com as funções declaradas no arquivo .h implementadas. Para que o código fonte possa ser reutilizado, é necessário chamar o arquivo .h dentro do novo código. Essa chamada é feita no seguinte formato: `#include "arquivo.h"`. A inclusão com aspas duplas é feita quando se trata de um arquivo .h no diretório corrente (dentro do projeto). No cabeçalho é mais fácil ver as funções disponíveis em comparação com o código c (imagine um com mais de um milhão de linhas). Os compiladores usam arquivos .h para fazer referência aos métodos usados no código do programa.

Programação modularizada: separação de partes distintas do código.

Tipo de dados, estrutura de dados, tipo abstrato de dados (TAD): termos parecidos, mas com significados diferentes. Veremos essas definições a seguir.

## Estrutura de Dados

Tipo de dados: estabelece o domínio de uma variável, como int, float, char, etc

Por exemplo, uma variável do tipo int pode ter apenas valores inteiros entre -2.147.483.648 e +2.147.483.647

Novos tipos de dados podem ser definidos em termos de outros já definidos, como registros, uniões, enumerados, vetores, matrizes.

Estrutura de dados: define um relacionamento lógico entre tipos de dados.

Conforme apresentada na Wikipédia: "estrutura de dados é um modo particular de armazenamento e organização de dados em um computador de modo que possam ser usados eficientemente, facilitando a busca e a modificação."

Estrutura de dados também é uma coleção de valores, considerando os seus relacionamentos e operações.

Uma estrutura pode ser:

- homogênea: vetores e matrizes de dados do tipo numérico (int, float, etc) ou caractere. Em outras palavras, esses conjuntos comportam apenas um único tipo de dado.
- heterogênea: engloba estruturas e uniões, que além de "agrupar" diferentes tipos de dados em um único tipo, também podem estar presentes em vetores e matrizes. Em outras palavras, ao declararmos uma struct, por exemplo, estamos definindo um "novo" tipo de dado. Assim, podemos também definir vetores e matrizes para essa estrutura.

→ Exemplo de vetor de estrutura: lista de cadastro de alunos

→ Exemplo de matriz de estrutura: uma imagem colorida no formato RGB (red green blue), onde cada elemento (pixel) contém três campos para representar a tonalidade de cada cor.

Exemplo de estrutura de dados: vetor (arranjo -- array) de inteiros (int):




- É uma estrutura linear, ou seja, possui uma única dimensão.
- O vetor pode ser estático ou dinâmico.
- Cada elemento do vetor é um número inteiro.
- A estrutura contém um número limitado (finito) de elementos: independentemente se é estático ou dinâmico, um vetor deve ter um tamanho definido.
- Fácil e rápido acesso aos elementos, podendo ser por índice ou aritmética de ponteiros
- Para a remoção de elementos, podem haver deslocamento de elementos, mas uma cópia do último elemento é mantida, pois não é possível desalocar apenas um único elemento em vetores.

As variáveis da estrutura juntas têm um significado. Por exemplo, representar alguns dados de disciplinas:

```
typedef struct{
    char nome_disc[101];
    int RA;
    float notas[3];
}NotaDisciplina;
```

### Tipo abstrato de dados (TAD)

Conjunto bem definido das estruturas e do grupo de operações que podem ser aplicados nelas

mun <u>do</u> real	dados de interesse	ESTRUTURA de armazenamento	possíveis OPERAÇÕES
 pessoa	<ul style="list-style-type: none"> <li>a idade da pessoa</li> </ul>	<ul style="list-style-type: none"> <li>tipo inteiro</li> </ul>	<ul style="list-style-type: none"> <li>nasce ( <math>i \leftarrow 0</math> )</li> <li>aniversário ( <math>i \leftarrow i + 1</math> )</li> </ul>
 cadastro de funcionários	<ul style="list-style-type: none"> <li>o nome, cargo e o salário de cada funcionário</li> </ul>	<ul style="list-style-type: none"> <li>tipo lista ordenada</li> </ul>	<ul style="list-style-type: none"> <li>entra na lista</li> <li>sai da lista</li> <li>altera o cargo</li> <li>altera o salário</li> </ul>
 fila de espera	<ul style="list-style-type: none"> <li>nome de cada pessoa e sua posição na fila</li> </ul>	<ul style="list-style-type: none"> <li>tipo fila</li> </ul>	<ul style="list-style-type: none"> <li>sai da fila (o primeiro)</li> <li>entra na fila (no fim)</li> </ul>

Estruturas de dados são utilizadas pelo programa através de operações apropriadas: o programador define o que pode ser feito para cada estrutura.

Muitas vezes é conveniente pensar nas operações suportadas por estruturas de dados em vez de na maneira como elas são implementadas: vocês já devem ter utilizados várias funções de várias bibliotecas para diversas tarefas. Para usar essas funções, vocês precisam saber dos detalhes da implementação? Geralmente não!

Assim, os dados armazenados podem ser manipulados apenas pelos operadores: os detalhes de representação e de implementação são ocultados e apenas as funcionalidades são conhecidas (e.g., sabemos que a função `strlen` da biblioteca `string` serve para calcular o tamanho de uma string, mas não precisamos saber como ela foi implementada para o seu uso).

Essas abstrações podemos fazer através de tipo abstrato de dados (TAD), que é uma especificação de estruturas de dados e as suas operações possíveis.

Em outras palavras, TAD = estrutura de dados + operações de manipulação da estrutura.

Em um TAD, os dados armazenados devem ser manipulados apenas pelos operadores:

- os detalhes de implementação (representação e funções) são ocultados, ou seja, apenas as funcionalidades são conhecidas.
- Encapsulamento: evita acesso indevido aos dados e funcionalidades.
- Em um TAD é permitido o acesso aos dados apenas nas operações, ou seja, os dados devem ser manipulados apenas através de funções.
- Por fim, um outra finalidade do TAD é possibilitar o reuso de suas funcionalidades em diferentes aplicações. Isso depende da flexibilidade do TAD, a qual envolve a modularização do código, que veremos daqui a pouco.

Operações mais comuns em TADs:

- Criação de estrutura
- Inclusão de um elemento
- Atualização de um elemento
- Remoção de um elemento
- Acesso a um elemento
- Impressão

Exemplo de TAD:

arquivos em C: `FILE *arq;`

Acesso aos dados de `arq` somente por funções de manipulação do tipo `FILE`

`fopen()`  
`fclose()`  
`fscanf()`  
etc

## Modularização

Toda a manipulação da estrutura é feita por meio de funções que interagem com ela. Para cadastrar um novo aluno, por exemplo, não deve ser feito o seguinte:

```
int main() {
    NotaDisciplina nd;
    char nome_disc[101];
    int RA;
    float notas[3];

    printf("Nome disciplina: ");
    scanf("%[^\n]s", nome_disc);
    printf("Numero matricula: ");
    scanf("%d", &RA);
    printf("Informe 3 notas do aluno: ");
    scanf("%f %f %f", &notas[0], &notas[1], &notas[2]);

    strcpy(nd.nome_disc, nome_disc);
    nd.RA = RA;
    nd.notas[0] = notas[0];
    nd.notas[1] = notas[1];
    nd.notas[2] = notas[2];
}
```

Deve-se criar uma função para cadastro, por exemplo, a função `cadastrar()`. Assim, sempre que for preciso cadastrar um aluno, esta função é chamada em vez de acessar a estrutura de forma direta:

```
NotaDisciplina cadastrar(char nome[], int RA, float notas[]){
    NotaDisciplina disc;
    ...
    return disc;
}
```

Outro exemplo seria a criação de um método para calcular notas.

**Manipular a estrutura através de funções possibilita o reuso de código. Sempre que for preciso fazer uma operação sobre a estrutura, basta chamar a função responsável.**

Essa forma de implementação (por funções) é denominada modularização.

**Aviso importante: por convenção, o arquivo de cabeçalho e de código fonte devem ter o mesmo nome, alterando apenas a extensão: `minha_implementacao.h` e `minha_implementacao.c`**

A modularização possibilita ocultar a implementação e facilita a manutenção e a reutilização do código-fonte.

Quem usa o TAD não precisa saber como a estrutura foi implementada. Para o isso, basta conhecer apenas as funcionalidades do TAD. Por exemplo: com a biblioteca `string.h`, diversas funções podem ser usadas: `strlen` (pegar o tamanho da string), `strcpy` (copiar o conteúdo de uma string em outra), `strcmp` (comparar duas strings), etc. O uso dessas funcionalidades é possível sem a necessidade do conhecimento dos detalhes de implementação da biblioteca.

Também, o programador pode criar o seu próprio TAD e disponibilizá-lo para outras pessoas usarem.

Arquivo `.h`:

- Protótipos das funções
- Typedefs
- Variáveis globais

Arquivo `.c`:

- Declaração dos tipos de dados (ou isso pode ser feito no arquivo `.h`)
- Implementação das funções

**`usa_tad.c`:** o arquivo que conterà a função `main` e usará as funcionalidades do TAD.

## Exemplo de TAD

Criar um TAD para trabalhar com pontos: definir tipo de dado e definir operações que poderão ser utilizadas.

No código abaixo temos a estrutura `Ponto`, que tem o objetivo de guardar as coordenadas de um ponto 2D. Existem também algumas funções para manipular a estrutura:

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

typedef struct ponto Ponto;
struct ponto {
    float x;
    float y;
};
```

```
Ponto cria_ponto(float x, float y) {
    Ponto p;
    p.x = x;
    p.y = y;
    return p;
}

void imprime_ponto(Ponto p) {
    printf("%.2f, %.2f\n", p.x, p.y);
}

float calcula_distancia(Ponto p1, Ponto p2) {
    float dx, dy;
    dx = p2.x - p1.x;
    dy = p2.y - p1.y;
    return sqrt(pow(dx, 2) + pow(dy, 2));
}

int main() {
    Ponto p1, p2;
    float d;

    p1 = cria_ponto(10, 20);
    p2 = cria_ponto(10, 30);
    imprime_ponto(p1);
    imprime_ponto(p2);

    d = calcula_distancia(p1, p2);
    printf("%.2f", d);
}
```

### Reorganização do código respeitando os conceitos de TADs.

Definição do ponto.h:

```
typedef struct ponto Ponto;

Ponto cria_ponto(float x, float y);

void imprime_ponto(Ponto p);

float distancia(Ponto p1, Ponto p2);
```

Definição do ponto.c:

```
#include <stdio.h>
#include <math.h>
#include "ponto.h"

struct ponto {
    float x;
    float y;
};

Ponto cria_ponto(float x, float y) {
    Ponto p;
    p.x = x;
    p.y = y;
    return p;
}

void imprime_ponto(Ponto p){
    printf("%.2f, %.2f\n", p.x, p.y);
}

float distancia(Ponto p1, Ponto p2) {
    float dx, dy, dt;
    dx = p1.x - p2.x;

    dy = p1.y - p2.y;
    dt = sqrt(pow(dx, 2) + pow(dy, 2));
    return dt;
}
```

Criar o arquivo principal:

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

main() {
    Ponto p1, p2;
    float d;

    p1 = cria_ponto(10, 20);
    p2 = cria_ponto(10, 30);
    imprime_ponto(p1);
    imprime_ponto(p2);

    d = distancia(p1, p2);
    printf("%.2f", d);
}
```

Ao compilar o arquivo principal, ocorrerá erros, pois o arquivo principal não reconhece Ponto como estrutura. Em outras palavras, não há como saber quanto espaço uma variável do tipo Ponto precisa.

Essa é justamente a ideia de que o usuário do TAD não deve conhecer ou ter acesso direto a estrutura. Isso se chama encapsulamento. Isso é feito para o usuário não possa fazer alterações inesperadas sobre a estrutura.

O que podemos fazer é utilizar ponteiros para estrutura. Assim precisaremos fazer uma alteração em nosso código:

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

int main() {
    Ponto *p1, *p2; // p1 e p2 agora são ponteiros
    float d;

    p1 = cria_ponto(10, 20);
    p2 = cria_ponto(10, 30);
    imprime_ponto(p1);
    imprime_ponto(p2);

    d = distancia(p1, p2);
    printf("%.2f", d);
}
```

Perceba que p1 e p2 agora são ponteiros do tipo Ponto. Se o programa for compilado novamente, os erros na declaração de p1 e p2 foram resolvidos, embora ainda existam outros erros.

Perceba que p1 e p2 recebem o retorno da função cria\_ponto(). Esta função retorna um tipo Ponto, mas p1 é um ponteiro e espera um endereço. Assim, temos uma incompatibilidade de tipos, que pode ser resolvida alterando a função cria\_ponto para que ela retorne um ponteiro.

Correções no código:

```
ponto.h

typedef struct ponto Ponto;

Ponto * cria_ponto(float x, float y);

void imprime_ponto(Ponto *p);

float distancia(Ponto *p1, Ponto *p2);

void libera_ponto(Ponto *p);
```



arquivo .c

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "ponto.h" // Funciona no codeblocks quando criarmos os arquivos dentro de um projeto

struct ponto {
    float x;
    float y;
};

Ponto * cria_ponto(float x, float y){
    Ponto *p = malloc(sizeof(Ponto));
    p->x = x;
    p->y = y;
    return p;
}

void imprime_ponto(Ponto * p){
    printf("%.2f, %.2f\n", p->x, p->y);
}

float distancia(Ponto *p1, Ponto *p2) {
    float dx, dy, dt;
    dx = p1->x - p2->x;
    dy = p1->y - p2->y;
    dt = sqrt(pow(dx, 2) + pow(dy, 2));
    return dt;
}

void libera_ponto(Ponto *p) {
    free(p);
}
```

## Referências

Prata, S. C++ Primer Plus. Waite Group Press, 1998.

Szwarcfiter, J.; Markenzon, L. Estruturas de Dados e Seus Algoritmos. LTC, 2010.

Tenenbaum, A.; Langsam, Y. Estruturas de Dados usando C. Pearson, 1995.