

## **Parte II**

# **Estruturas Dinâmicas**

Nesta segunda parte do livro, apresentamos as estruturas de dados que convencionamos chamar de *dinâmicas*, pois oferecem suporte adequado para inserção e remoção de elementos dinamicamente. São estruturas em que o número de elementos armazenados não está limitado a uma dimensão inicial. As estruturas crescem em espaço de memória à medida que novos elementos são inseridos. Com isso, o número de elementos que podemos armazenar nessas estruturas é arbitrário.

No Capítulo 12, iniciamos esta parte do livro com a apresentação do conceito de *tipo abstrato de dados*, fundamental para o desenvolvimento de programas corretos e de fácil manutenção. Um tipo abstrato de dados encapsula (esconde) de quem o usa a forma concreta com que o tipo foi implementado. Este conceito é então usado em todos os capítulos subsequentes. No Capítulo 13, discutimos a implementação de *vetores dinâmicos* (e cadeias de caracteres dinâmicas) com o conceito de realocação de memória. Em seguida, no Capítulo 14, apresentamos as estruturas de *listas encadeadas*, que são amplamente utilizadas na elaboração de programas. Em seguida, são apresentadas suas variações: lista duplamente encadeada e lista circular. O Capítulo 15 apresenta as estruturas especializadas, *pilha* e *fila*, e suas implementações com vetores e listas encadeadas. Em seguida, no Capítulo 16, apresentamos as *árvores*, para representação de hierarquias. O capítulo discute a implementação de árvores binárias, incluindo árvores binárias de busca, e árvores com número de filhos variável. Por fim, no Capítulo 17, apresentamos técnicas de programação que permitem a implementação de *estruturas genéricas*, isto é, estruturas que podem ser usadas para armazenar qualquer tipo de dado.

## 12.1 Nódulos e manipulação em vetor

As estruturas dinâmicas que já vimos até aqui são estruturas compostas por nódulos. Um nódulo é um bloco de memória que armazena uma informação e que pode ser movido para outro bloco de memória. Isto é, é possível que o mesmo bloco de memória contenha diferentes informações ao longo do tempo. A maneira mais comum de se manipular nódulos é através de vetores.

Um vetor é uma estrutura de dados que armazena uma coleção de elementos. Cada elemento é identificado por um índice, que indica sua posição no vetor. Os vetores são muito usados para armazenar estruturas dinâmicas, como listas encadeadas e pilhas.

# Capítulo 12

## Tipo abstrato de dados

Nos capítulos anteriores, apresentamos a sintaxe da linguagem C para a criação e manipulação de tipos estruturados. Neste capítulo, discutiremos uma importante técnica de programação baseada na definição de tipos estruturados, conhecida como *tipo abstrato de dados* (TAD). A ideia central é encapsular (esconder) de quem usa determinado tipo a forma concreta com que o tipo foi implementado. Por exemplo, se criamos um tipo para representar um ponto no espaço, um cliente deste tipo usa-o de forma abstrata, baseando-se apenas nas funcionalidades oferecidas pelo tipo. A forma com que o tipo foi efetivamente implementado (armazenando cada coordenada num campo ou agrupando todas num vetor) passa a ser um detalhe de implementação, que não deve afetar o uso do tipo nos mais diversos contextos. Com isso, desacoplamos a implementação do uso, facilitando a manutenção e aumentando o potencial de reutilização do tipo criado. Por exemplo, a implementação do tipo pode ser alterada sem afetar o seu uso em outros contextos.

Veremos como a linguagem C pode nos ajudar na implementação de um TAD, através de alguns de seus mecanismos básicos de modularização, isto é, divisão de um programa em vários arquivos fontes.

### 12.1 Módulos e compilação em separado

Mencionamos que um programa em C pode ser dividido em vários arquivos fontes (arquivos com extensão “.c”). De fato, quando desenvolvemos programas, procuramos identificar funções afins e agrupá-las por arquivo. Quando temos um arquivo com funções que representam apenas parte da implementação de um programa completo, denominamos esse arquivo de *módulo*. Assim, a implementação de um programa pode ser composta por um ou mais módulos.

No caso de um programa composto por vários módulos, cada um desses módulos deve ser compilado separadamente, gerando um arquivo objeto (geralmente um arquivo com extensão “.o” ou “.obj”) para cada módulo. Após a compilação de todos os módulos, outra ferramenta, denominada *ligador* (*linker*), é usada para juntar todos os arquivos objetos em um único arquivo executável. É também na ligação dos objetos que os códigos objetos das funções da biblioteca padrão de C são incluídos no código objeto. Portanto, mesmo que nosso programa seja constituído por um único módulo, a ligação ocorre.

Para programas pequenos, o uso de vários módulos pode não se justificar. Mas, para programas de médio e grande porte, a sua divisão em vários módulos é mandatória, pois facilita a divisão de uma tarefa maior e mais complexa em tarefas menores e, provavelmente, mais fáceis de implementar e de testar. Além disso, um módulo com funções C pode ser reutilizado para compor vários programas e, assim, poupar muito tempo de reprogramação.

Para ilustrar o uso de módulos em C, vamos considerar a existência de um arquivo “str.c” que contém apenas a implementação das funções de manipulação de *strings* vistas no Capítulo 8 ( `comprimento` , `copia` e `concatena` ). Considere também que temos um arquivo “prog1.c” com o seguinte código:

```
#include <stdio.h>

int comprimento (char* str);
void copia (char* dest, char* orig);
void concatena (char* dest, char* orig);

int main (void)
{
    char str[101], str1[51], str2[51];
    printf("Digite uma sequência de caracteres: ");
    scanf(" %50[^\\n]", str1);
    printf("Digite outra sequência de caracteres: ");
    scanf(" %50[^\\n]", str2);
    copia(str, str1);
    concatena(str, str2);
    printf("Comprimento da concatenação: %d\\n", comprimento(str));
    return 0;
}
```

A partir desses dois arquivos fontes, podemos gerar um programa executável compilando cada um dos arquivos separadamente e depois ligando-os em um único arquivo executável. Isto é feito automaticamente quando criamos um projeto em um IDE (ambiente integrado de desenvolvimento) e pedimos para o executável ser gerado.

O mesmo arquivo “str.c” pode ser usado para compor outros programas que queiram utilizar suas funções. Para que as funções implementadas em “str.c” possam ser usadas por outro módulo C, este precisa conhecer os protótipos das funções oferecidas. No código mostrado de “prog1.c”, isso foi resolvido através da inclusão dos protótipos das funções no início do arquivo. Entretanto, para módulos que ofereçam várias funções ou que queiram usar funções de muitos outros módulos, essa inclusão manual pode ficar muito trabalhosa e sensível a erros. Para contornar esse problema, todo módulo de funções C costuma ter associado um arquivo que contém apenas os protótipos das funções oferecidas pelo módulo e, eventualmente, as definições e os tipos de dados exportados ( `define`’s, `typedef`’s etc.). Esse arquivo de protótipos caracteriza a *interface do módulo* e, em geral, segue o mesmo nome do módulo ao qual está associado, só que com a extensão “.h”. Assim, podemos definir um arquivo de interface “str.h” para o módulo do exemplo anterior, com o seguinte conteúdo:

```

/* Funções oferecidas pelo módulo str */

/* Função comprimento
** Retorna o número de caracteres da string passada como parâmetro
*/
int comprimento (char* str);

/* Função copia
** Copia os caracteres da string orig (origem) para dest (destino)
*/
void copia (char* dest, char* orig);

/* Função concatena
** Concatena a string orig (origem) na string dest (destino)
*/
void concatena (char* dest, char* orig);

```

Observe que colocamos vários comentários no arquivo “str.h”. Isso é uma prática muito comum, e tem como finalidade documentar as funções oferecidas por um módulo. Esses comentários devem esclarecer qual é o comportamento esperado das funções exportadas pelo módulo, facilitando o seu uso por outros programadores (ou pelo mesmo programador algum tempo depois da criação do módulo).

Agora, em vez de incluir manualmente os protótipos dessas funções, todo módulo que quiser usar as funções de “str.c” precisa apenas incluir seu arquivo de interface “str.h”. No exemplo anterior, o código em “prog1.c” seria então escrito da seguinte forma:

```

#include <stdio.h>
#include "str.h"

int main (void)
{
    char str[101], str1[51], str2[51];

    printf("Digite uma sequência de caracteres: ");
    scanf(" %50[^\\n]", str1);
    printf("Digite outra sequência de caracteres: ");
    scanf(" %50[^\\n]", str2);
    copia(str, str1);
    concatena(str, str2);

    printf("Comprimento da concatenação: %d\\n", comprimento(str));
    return 0;
}

```

Note que os arquivos de interface da biblioteca padrão de C (que acompanham seu compilador) são incluídos da forma `# include < arquivo .h >`, enquanto os arquivos de interface dos nossos módulos são geralmente incluídos da forma `# include " arquivo .h "`, conforme foi discutido na Seção 4.7.

## 12.2 Tipo abstrato de dados

Geralmente, um módulo agrupa tipos e funções com funcionalidades relacionadas, caracterizando assim uma finalidade bem definida. Por exemplo, na seção anterior vimos um módulo com funções para a manipulação de cadeias de caracteres. Nos casos em que um módulo define um novo tipo de dado e o conjunto de operações para manipular dados desse tipo, dizemos que o módulo representa um *tipo abstrato de dados* (TAD). Nesse contexto, *abstrato* significa “abstraída a forma de implementação”, ou seja, um TAD é descrito pela finalidade do tipo e de suas operações, e não pela forma como está implementado.

Podemos, por exemplo, criar um TAD para representar matrizes alocadas dinamicamente. Para isso, criamos um tipo “matriz” e uma série de funções que o manipulam. Podemos pensar, por exemplo, em funções que acessam e manipulam os valores dos elementos da matriz. Criando um tipo abstrato, podemos “esconder” a estratégia de implementação. Quem usa o tipo abstrato precisa apenas conhecer a funcionalidade que ele implementa, não a forma como ele é implementado. Isto facilita a manutenção e a reutilização de códigos.

A divisão de programas em módulos e a criação de TADs são técnicas de programação muito importantes. Nos próximos capítulos, vamos procurar dividir nossos exemplos e programas em módulos e usar tipos abstratos de dados, sempre que isso for possível. Antes, porém, vamos ver alguns exemplos completos de TADs.

A interface de um TAD consiste, basicamente, na definição do nome do tipo e do conjunto de funções exportadas para sua criação e manipulação. É comum tipos distintos oferecerem operações similares. Por exemplo, é fácil imaginar que qualquer tipo abstrato em geral oferecerá uma função para sua criação – mais precisamente, para a criação de instâncias do tipo. Para permitir que tipos distintos sejam usados por um único cliente (situação muito comum em aplicações reais), precedemos os nomes das funções exportadas por um prefixo que identifica a qual tipo as funções se aplicam. Por exemplo, a função para criar um `Ponto` pode ser chamada de `pto_cria`, enquanto a função para criar um `Circulo` pode se chamar `circ_cria`. Dessa forma, funções para criar tipos distintos terão nomes distintos e poderão ser usadas dentro de um mesmo contexto. Se não aplicássemos essa regra, provavelmente teríamos funções de mesmo nome sendo exportadas por tipos distintos, o que inviabilizaria a utilização dos tipos simultaneamente, pois haveria duplicação de símbolos (um mesmo nome usado para identificar duas funções distintas).

Portanto, recomendamos utilizar um prefixo nos nomes das funções exportadas pelo módulo. Se optarmos por utilizar variáveis globais e funções auxiliares na implementação dos módulos, elas serão declaradas como estáticas, sendo assim visíveis apenas dentro do arquivo que implementa o módulo.

Se optarmos por exportar variáveis globais, a declaração da variável na interface é precedida pelo modificador `extern`. Por exemplo, se a variável global `int tst_valor`; é declarada em um módulo “`teste.c`”, a declaração `extern int tst_valor`; deve existir na interface “`teste.h`”. No entanto, como já dissemos, devemos evitar o uso de variáveis globais, em especial, devemos evitar a exportação de variáveis globais em um módulo.

### 12.2.1 Exemplo 1: TAD Ponto

Como primeiro exemplo de TAD, vamos considerar a criação de um tipo de dado para representar um ponto no  $\mathbb{R}^2$ . Para isso, devemos definir um tipo abstrato, que denomi-

naremos Ponto, e o conjunto de funções que operam sobre esse tipo. Neste exemplo, vamos considerar as seguintes operações:

- **cria**: operação que cria um ponto com coordenadas  $(x, y)$ .
- **libera**: operação que libera a memória alocada por um ponto.
- **acessa**: operação que retorna as coordenadas de um ponto.
- **atribui**: operação que atribui novos valores às coordenadas de um ponto.
- **distancia**: operação que calcula a distância entre dois pontos.

A interface desse módulo pode ser dada pelo arquivo “ponto.h” ilustrado a seguir:

```
/* TAD: Ponto (x, y) */

/* Tipo exportado */
typedef struct ponto Ponto;

/* Funções exportadas */

/* Função cria
** Aloca e retorna um ponto com coordenadas (x, y)
*/
Ponto* pto_cria (float x, float y);

/* Função libera
** Libera a memória de um ponto previamente criado
*/
void pto_libera (Ponto* p);

/* Função acessa
** Retorna os valores das coordenadas de um ponto
*/
void pto_acessa (Ponto* p, float* x, float* y);

/* Função atribui
** Atribui novos valores às coordenadas de um ponto
*/
void pto_atribui (Ponto* p, float x, float y);

/* Função distancia
** Retorna a distância entre dois pontos
*/
float pto_distancia (Ponto* p1, Ponto* p2);
```

Note que a composição da estrutura Ponto ( `struct ponto` ) não é exportada pelo módulo, isto é, não faz parte da interface do módulo e, portanto, não é visível para outros módulos. Dessa forma, os demais módulos que usarem esse TAD não poderão acessar diretamente os campos dessa estrutura. Os clientes desse TAD só terão acesso às informações obtidas através das funções exportadas pelo arquivo “ponto.h”.

Conhecendo apenas a interface do TAD, podemos criar programas que façam uso das funcionalidades exportadas. O arquivo que usa o TAD deve, obrigatoriamente, incluir o arquivo que define sua interface. Por exemplo:

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

int main (void)
{
    Ponto* p = pto_cria(2.0,1.0);
    Ponto* q = pto_cria(3.4,2.1);

    float d = pto_distancia(p,q);

    printf("Distancia entre pontos: %f\n",d);

    pto_libera(q);
    pto_libera(p);

    return 0;
}
```

Logicamente, precisamos ligar o arquivo com a implementação do módulo para gerar um executável. No entanto, salientamos mais uma vez que a forma da implementação não deve alterar o uso do tipo abstrato, isto é, podemos alterar a implementação do módulo mantendo o código funcionando sem nenhuma modificação.

Agora, mostraremos uma implementação para esse tipo abstrato de dados. O arquivo de implementação do módulo (arquivo “ponto.c”) deve sempre incluir o arquivo de interface do módulo. Isto é necessário por duas razões. Primeiro, podem existir definições na interface que são necessárias na implementação. No nosso caso, por exemplo, precisamos da definição do tipo `Ponto`. A segunda razão é garantir que as funções implementadas correspondem às funções da interface. Como os protótipos das funções exportadas são incluídos, o compilador verifica, por exemplo, se os parâmetros das funções implementadas equivalem aos parâmetros dos protótipos. Além da própria interface, precisamos, naturalmente, incluir as interfaces das funções que usamos da biblioteca padrão.

```
#include "ponto.h"

#include <stdlib.h> /* malloc, free, exit */
#include <stdio.h> /* printf */
#include <math.h> /* sqrt */
```

Como só precisamos guardar as coordenadas de um ponto, podemos definir a estrutura `ponto` da seguinte forma:

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

A função que cria um ponto dinamicamente deve alocar a estrutura que representa o ponto e inicializar os seus campos:

```
Ponto* pto_cria (float x, float y)
{
    Ponto* p = (Ponto*) malloc(sizeof(Ponto));
    if (p == NULL) {
        printf("Memória insuficiente!\n");
        exit(1);
    }
    p->x = x;
    p->y = y;
    return p;
}
```

Para esse TAD, a função que libera um ponto deve apenas liberar a estrutura que foi criada dinamicamente através da função cria:

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

As funções para acessar e atribuir valores às coordenadas de um ponto são de fácil implementação, como pode ser visto a seguir. Essas funções permitem que uma função cliente tenha acesso às coordenadas do ponto sem conhecer a forma concreta pela qual estes valores são armazenados na estrutura que representa o tipo. Uma possível implementação dessas funções é:

```
void pto_acessa (Ponto* p, float* x, float* y)
{
    *x = p->x;
    *y = p->y;
}

void pto_atribui (Ponto* p, float x, float y)
{
    p->x = x;
    p->y = y;
}
```

Já a operação para calcular a distância entre dois pontos pode ser implementada da seguinte forma:

```
float pto_distancia (Ponto* p1, Ponto* p2)
{
    float dx = p2->x - p1->x;
    float dy = p2->y - p1->y;
    return sqrt(dx*dx + dy*dy);
}
```

A junção destes fragmentos de código formam o conteúdo do arquivo de implementação do módulo.

### 12.2.2 Exemplo 2: TAD Círculo

Podemos aproveitar o tipo estruturado que representa um círculo, visto anteriormente, e implementar um tipo abstrato de dados. As seguintes operações podem ser oferecidas:

- **cria**: operação que cria um círculo com centro  $(x,y)$  e raio  $r$ .
- **libera**: operação que libera a memória alocada por um círculo.
- **area**: operação que calcula a área do círculo.
- **interior**: operação que verifica se um dado ponto está dentro do círculo.

A interface desse TAD pode ser dada pelo arquivo “circulo.h” apresentado a seguir. Nos exemplos que se seguem, omitiremos os comentários dos arquivos de interface para que o texto fique mais conciso; no entanto, em aplicações reais, recomendamos fortemente a inclusão de uma documentação adequada, na forma de comentários, nos arquivos de interface.

```
/* TAD: Círculo */

/* Dependência de módulos */
#include "ponto.h"

/* Tipo exportado */
typedef struct circulo Circulo;

/* Funções exportadas */
/* Função cria
** Aloca e retorna um círculo com centro (x, y) e raio r
*/
Circulo* circ_cria (float x, float y, float r);

/* Função libera
** Libera a memória de um círculo previamente criado
*/
void circ_libera (Circulo* c);

/* Função area
** Retorna o valor da área do círculo
*/
float circ_area (Circulo* c);

/* Função interior
** Verifica se um dado ponto p está dentro do círculo
*/
int circ_interior (Circulo* c, Ponto* p);
```

Devemos notar que a operação **interior** faz uso do tipo **Ponto**, portanto a interface “ponto.h” foi incluída na interface do tipo **Circulo**. Esta dependência entre módulos é muito comum. Agora, vamos imaginar um programa que faz uso tanto do TAD **Ponto** como do TAD **Circulo**. Este código iria incluir os arquivos de interface de ambos os TAD’s, pois o cliente não necessariamente sabe que a interface do círculo inclui a

interface do ponto. Incluir um arquivo duas ou mais vezes não necessariamente cria erros de compilação, mas algumas definições em interfaces não podem ser duplicadas (por exemplo, a definição de macros). De qualquer forma, incluir mais de uma vez um arquivo de interface aumenta o tempo de compilação. Para evitar eventuais erros e diminuir o tempo de compilação do código, em geral, “cercamos” o código da interface com diretivas de pré-compilação. Por exemplo, o arquivo “ponto.h” pode ser dado por:

```
#ifndef PONTO_H
#define PONTO_H
...
/* código da interface do módulo */
#endif
```

Com isso, na primeira vez que o arquivo for processado pelo compilador, o símbolo `PONTO_H` não estará definido. Então, o símbolo é definido e o conteúdo do arquivo considerado. Se o compilador processar este mesmo arquivo uma segunda vez para a compilação de um módulo, o símbolo `PONTO_H` já estará definido e o conteúdo do arquivo não será considerado pelo compilador, pois a diretiva `#ifndef` resultará em falso e o código até o `#endif` correspondente ficará desabilitado. O nome do símbolo (`PONTO_H`, no caso) pode ser qualquer. Em geral, usamos o próprio nome do arquivo para criação do nome do símbolo. Analogamente, o arquivo “circulo.h” seria:

```
#ifndef CIRCULO_H
#define CIRCULO_H
...
/* código da interface do módulo */
#endif
```

Também podemos usar a diretiva `#if` para alternar entre trechos de código:

```
#if 0
    ...
    /* trecho de código desabilitado (não considerado) */
#else
    ...
    /* trecho de código habilitado */
#endif
```

Naturalmente, se trocarmos `#if 0` por `#if 1` alternamos a visibilidade dos trechos de códigos. Esta diretiva também é muito usada para ligar/desligar um trecho de código, sem incluir a cláusula `#else`.

Uma possível implementação do tipo `Circulo`, arquivo “circulo.c”, é apresentada a seguir. Salientamos a existência de um TAD Ponto na representação do círculo. Isto é, o uso deste módulo exige o uso do TAD Ponto.

```
#include <stdlib.h>
#include "circulo.h"
#define PI 3.14159

struct circulo {
    Ponto* p;
    float r;
};
```

```

Circulo* circ_cria (float x, float y, float r)
{
    Circulo* c = (Circulo*)malloc(sizeof(Circulo));
    c->p = pto_cria(x,y);
    c->r = r;
    return c;
}

void circ_libera (Circulo* c)
{
    pto_libera(c->p);
    free(c);
}

float circ_area (Circulo* c)
{
    return PI*c->r*c->r;
}

int circ_interior (Circulo* c, Ponto* p)
{
    float d = pto_distancia(c->p,p);
    return (d<c->r);
}

```

### 12.2.3 Exemplo 3: TAD Matriz

Como a implementação de um TAD fica “escondida” dentro de seu módulo, podemos experimentar diferentes maneiras de implementar um mesmo TAD, sem que isso afete os clientes. Para ilustrar essa independência de implementação, vamos considerar a criação de um tipo abstrato de dados para representar matrizes de valores reais alocadas dinamicamente, com dimensões  $m$  por  $n$  fornecidas em tempo de execução. Para tanto, devemos definir um tipo abstrato, que denominaremos **Matriz**, e o conjunto de funções que operam sobre esse tipo. Neste exemplo, vamos considerar as seguintes operações:

- **cria**: operação que cria uma matriz de dimensão  $m$  por  $n$ .
- **libera**: operação que libera a memória alocada para a matriz.
- **acessa**: operação que acessa o elemento da linha  $i$  e da coluna  $j$  da matriz.
- **atribui**: operação que atribui o elemento da linha  $i$  e da coluna  $j$  da matriz.
- **linhas**: operação que retorna o número de linhas da matriz.
- **colunas**: operação que retorna o número de colunas da matriz.

A interface do módulo, arquivo “matriz.h”, pode ser dada pelo código a seguir:<sup>1</sup>

```
/* TAD: matriz m por n */
#ifndef MATRIZ_H
#define MATRIZ_H
typedef struct matriz Matriz;
Matriz* mat_cria (int m, int n);
void mat_libera (Matriz* mat);
float mat_acessa (Matriz* mat, int i, int j);
void mat_atribui (Matriz* mat, int i, int j, float v);
int mat_linhas (Matriz* mat);
int mat_colunas (Matriz* mat);
#endif
```

Conforme discutimos na Seção 7.3, a implementação de uma matriz alocada dinamicamente pode ser feita usando duas estratégias distintas: matrizes dinâmicas representadas por vetores simples e matrizes dinâmicas representadas por vetores de ponteiros. A interface do módulo independe da estratégia de implementação adotada, o que é altamente desejável, pois podemos mudar a implementação sem afetar as aplicações que fazem uso do tipo abstrato. Se usarmos a estratégia com vetores simples, a estrutura que representa a matriz pode ser definida por:

```
struct matriz {
    int lin;
    int col;
    float* v;
};
```

Se usarmos a estratégia com vetores de ponteiros, a estrutura pode ser dada por:

```
struct matriz {
    int lin;
    int col;
    float** v;
};
```

Fica como exercício a implementação das funções usando as duas estratégias alternativas. Independentemente da estratégia utilizada, a funcionalidade oferecida pelo tipo abstrato não se altera.

<sup>1</sup>Nos demais exemplos, omitiremos o “cerco” com diretivas de pré-processamento para o texto ficar conciso; no entanto, recomendamos o uso das diretivas em todos os arquivos de interface.

## Exercícios

1. Acrescente novas operações ao TAD ponto, de tal forma que seja possível obter uma representação do ponto em coordenadas polares.
2. Usando apenas as operações definidas pelo TAD **Matriz**, implemente uma função que, dada uma matriz, crie dinamicamente a matriz transposta correspondente.
3. Defina a interface e implemente um TAD para representar números complexos. Sabe-se que um número complexo é representado por  $a + bi$ , onde  $a$  e  $b$  são números reais e  $i$  a unidade imaginária. O TAD deve implementar as seguintes operações:
  - (a) Função para criar um número complexo, dados  $a$  e  $b$ .
  - (b) Função para liberar um número complexo previamente criado.
  - (c) Função para somar dois números complexos, retornando um novo número com o resultado da operação. Sabe-se que:

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

- (d) Função para subtrair dois números complexos, retornando um novo número com o resultado da operação. Sabe-se que:

$$(a + bi) - (c + di) = (a - c) + (b - d)i$$

- (e) Função para multiplicar dois números complexos, retornando um novo número com o resultado da operação. Sabe-se que:

$$(a + bi)(c + di) = (ac - bd) + (bc + ad)i$$

- (f) Função para dividir dois números complexos, retornando um novo número com o resultado da operação. Sabe-se que:

$$\frac{a + bi}{c + di} = \left( \frac{ac + bd}{c^2 + d^2} \right) + \left( \frac{bc - ad}{c^2 + d^2} \right) i$$

Escreva um programa que use números complexos para testar sua implementação.