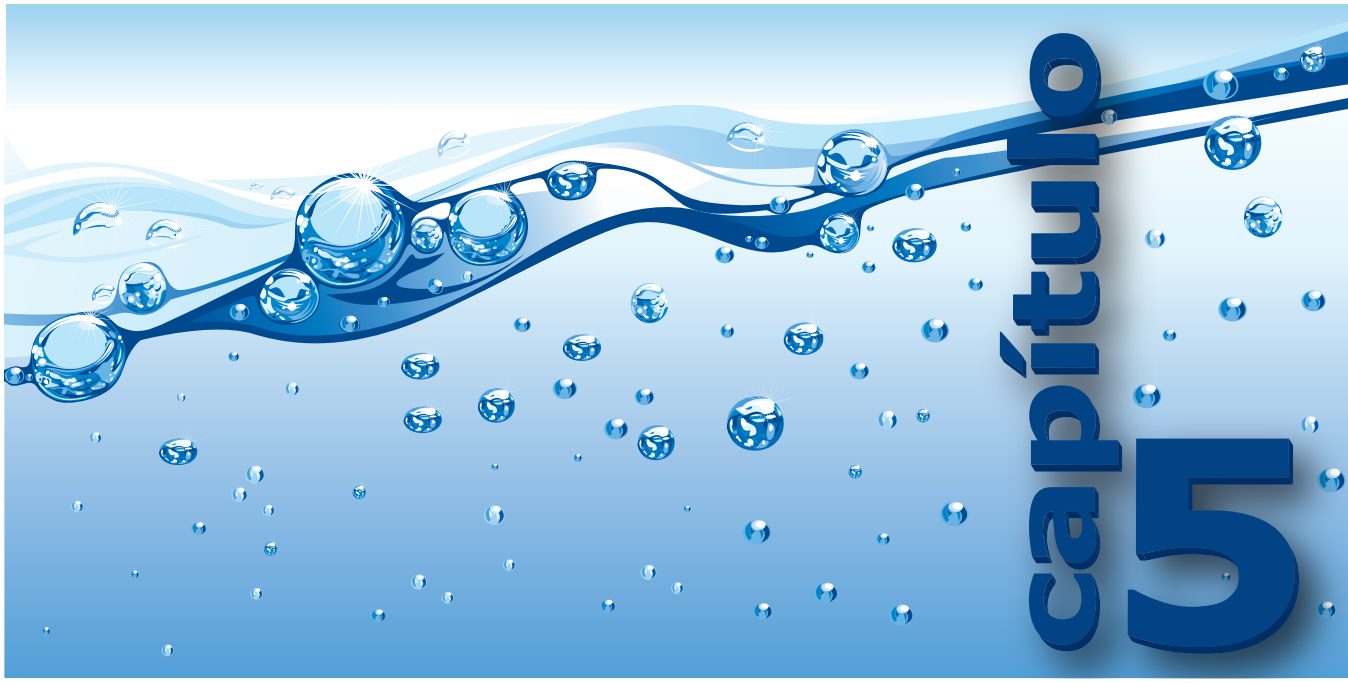

Conceitos Básicos de Estruturas de Dados

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar os seguintes conceitos:
  - ❒ Tipo de dado
  - ❒ Tipo primitivo
  - ❒ Tipo derivado
  - ❒ Tipo estruturado
  - ❒ Abstração
  - ❒ Implementação
  - ❒ Estrutura de dados
  - ❒ Tipo opaco
  - ❒ Tipo transparente
  - ❒ Algoritmo
  - ❒ Caso de entrada
  - ❒ Equivalência funcional
  - ❒ Refinamentos sucessivos
  - ❒ Linguagem algorítmica
  - ❒ Pseudolinguagem
  - ❒ Pseudocódigo
  - ❒ Paradigma algorítmico
  - ❒ Ponteiro opaco
  - ❒ Cliente de tipo de dado
  - ❒ Ocultação de informação
  - ❒ Abstração de alto nível
  - ❒ Encapsulamento de dados
  - ❒ Tipo abstrato de dado (TAD)
  - ❒ Variáveis estruturadas homogênea e heterogênea
  - ❒ Abstração de baixo nível
- Descrever o que se estuda na disciplina Estruturas de Dados
- Defender a importância da organização de dados na construção de um programa
- Classificar tipos de dados estruturados
- Fazer distinção entre algoritmo e receita culinária
- Expressar a relação entre organização de dados e algoritmo
- Descrever as etapas envolvidas na escrita de um algoritmo
- Testar um algoritmo
- Implementar um TAD em C

objetivos

INFORMAÇÕES USADAS NO COTIDIANO podem ser organizadas de diversas maneiras. Algumas dessas maneiras facilitam a vida, enquanto outras nem tanto. Considere, por exemplo, as várias maneiras com que se podem organizar uma lista de compras de supermercado. Tal lista pode ser organizada em ordem alfabética de itens, o que, você concordará, não é muito útil, pois dificilmente encontrará um supermercado no qual os itens são organizados desse modo. Uma maneira mais útil de organizar uma lista de compras é ordenando-a de acordo com a ordem de visitação de seções no supermercado que você frequenta com assiduidade. O problema agora é que, se você passar a frequentar outro supermercado, essa forma de organização de lista poderá perder muito de sua utilidade.

A discussão sobre como organizar uma lista de compras poderia se prolongar, mas o objetivo aqui é fazer uma analogia entre as facilidades para um consumidor determinadas pelo modo como se organiza uma lista de compras e a facilidade para um programador estabelecida pelo modo como ele organiza os dados que seu programa irá processar.

Os dados processados por um programa podem ser organizados de maneiras bem distintas. Algumas formas de organização de dados ocupam mais espaço do que outras, mesmo que representem os mesmos dados. Além disso, as mesmas operações sobre os mesmos dados podem ser mais ou menos eficientes dependendo do modo como eles são organizados. Assim organizar adequadamente os dados que serão processados por um programa é um requisito fundamental em seu processo de desenvolvimento do programa, porque, uma vez que os dados de um programa tenham sido convenientemente organizados, a escrita dos algoritmos que irão processá-los pode ser facilitada.

Este capítulo apresenta conceitos fundamentais de algoritmos e estruturas de dados. Em especial, ele explora os conceitos de tipos de dados transparentes e opacos (TADs) e mostra como implementá-los em C.

5.1 Definições Fundamentais

A disciplina **Estruturas de Dados** ensina como dados podem ser organizados, armazenados e processados. Mas, fora do contexto didático, uma estrutura de dados refere-se a uma forma sistemática de organização e acesso a dados, como será visto mais adiante.

Um **tipo de dado** (ou, simplesmente, **tipo**) consiste num conjunto de valores munido de uma coleção de operações permitidas sobre eles. Um **tipo primitivo** (ou **embutido**) é um tipo incorporado numa linguagem de programação e representado por uma palavra-chave. Linguagens de programação oferecem ainda a possibilidade de criação de outros tipos de dados. Esses tipos, denominados **tipos derivados**, podem ser criados pelo próprio programador ou são providos por bibliotecas.

Uma **variável estruturada** (ou **agregada**) é aquela que contém componentes que podem ser acessados individualmente. Uma variável estruturada pode ser **homogênea**, quando seus componentes são todos de um mesmo tipo, ou **heterogênea**, quando seus componentes podem ser de tipos diferentes. O tipo de uma variável estruturada é considerado um **tipo estruturado** (ou **agregado**). Em C, arrays são exemplos de variáveis estruturadas homogêneas (v. **Seção 3.1**), ao passo que estruturas e uniões (v. **Seção 3.10**) constituem exemplos de variáveis estruturadas heterogêneas.

Um tipo de dado é um conceito abstrato envolvendo um conjunto de propriedades. Uma vez que um tipo de dado tenha sido definido e as operações envolvendo objetos desse tipo tenham sido especificadas, pode-se implementá-lo. Uma implementação pode ser obtida via hardware ou software.

Um **cliente** (ou **programa-cliente**) de um tipo de dado é um programa que utiliza variáveis e operações sobre variáveis desse tipo. Neste livro, *cliente* sempre tem esse significado ou refere-se a um programador que cria programas-clientes.

Em muitas linguagens de programação, toda variável precisa ser definida antes de ser usada. Uma **definição de variável** tem três objetivos:

[1] Prover uma interpretação para o espaço em memória ocupado pela variável

[2] Fazer com que seja alocado espaço suficiente para conter a variável

[3] Associar esse espaço a um identificador (i.e., o nome da variável)

Em programação, pensa-se sobre tipos de dados primitivos em termos daquilo que se pode fazer com eles e não como eles são representados. Por exemplo, quando se lida com o tipo **int** de C não se está preocupado em como valores desse tipo são representados em memória ou como a operação de soma de valores desse tipo é implementada. Por exemplo, se alguém deseja somar dois inteiros, certamente, não estará preocupada nos detalhes do mecanismo pelo qual o resultado será obtido. Aqui, a pessoa está interessada no conceito matemático (abstrato) de inteiros e não na manipulação de bits.

Encapsulamento de dados consiste numa separação entre a representação de dados e as aplicações que os usam num nível conceitual por meio de **ocultação de informação**, que faz com que o usuário dos dados (programador) não enxergue a implementação; ele lida com os dados por meio de uma interface lógica. A vantagem da ocultação de informação para o programador reside no fato de ele não precisar se preocupar com detalhes de implementação nem usar o tipo indevidamente de modo que possa introduzir bugs no programa. Encapsulamento de dados permite definições de tipos de dados similares aos tipos primitivos de uma linguagem de programação de alto nível, como, por exemplo, o tipo **int** da linguagem C. Ocultar a representação de um tipo assegura consistência em seu uso e menor custo de manutenção dos programas que o utilizarem se a representação for modificada posteriormente. Encapsulamento de dados ou ocultação informações faz com que um *tipo de dado* seja um *tipo abstrato de dado* (v. **Seção 5.2**).

5.2 Tipos Abstratos e Estruturas de Dados

Uma **abstração de dado** consiste numa separação entre as propriedades conceituais de um tipo de dado e sua **implementação**. Por exemplo, inteiros podem ser representados de diversas maneiras (p. ex., BCD, complemento de dois) e o fato de o programador desconhecer essa representação não o impede de usar inteiros. Ou seja, o que o programador precisa saber é apenas como definir e efetuar operações com inteiros.

O conceito de abstração também é importante no cotidiano. Por exemplo, pode-se entender o conceito de freio de automóvel sem levar em consideração se trata-se de um freio a disco, a tambor ou ABS. Isto é, o conceito (abstração) de freio é independente de sua implementação. A separação entre abstração, que diz respeito ao uso de um sistema, e os detalhes de implementação, que dizem respeito a como o sistema é implementado, facilita o entendimento de sistemas complexos.

Um **tipo abstrato de dado** (abreviadamente, **TAD**) é um tipo cujas propriedades (domínio e operações) são especificadas independentemente de qualquer implementação. Um tipo abstrato de dado é um tipo de dado que não permite que sua forma de representação de dados seja disponível para clientes (v. **Seção 5.1**). Mais precisamente, um tipo abstrato de dado permite que o programador defina variáveis desse tipo, mas o acesso aos dados constituintes dessas variáveis é permitido apenas por meio da interface do tipo (i.e., pelas operações que são públicas). Como a representação de dados não é exposta, essa representação e as operações que a manipulam podem ser alteradas sem afetar os clientes.

De acordo com a última definição, todos os tipos primitivos de C são tipos abstratos de dados. Por exemplo, os clientes de um tipo inteiro de C não têm acesso à representação desse tipo, de modo que ela pode ser alterada sem afetar as operações com inteiros realizadas por algum cliente do tipo. Por outro lado, em C, strings não constituem um tipo abstrato de dado, pois programadores dessa linguagem sabem que strings são representados

por arrays de caracteres terminados pelo caractere nulo e usam esse conhecimento para processá-los. Assim a representação de strings não pode ser alterada sem que muitos programas que a utilizam deixem de funcionar.

Na construção de um tipo abstrato de dado, é importante que existam facilidades para ocultar detalhes de implementação (representação do tipo) dos programadores que são clientes do tipo.

Um tipo abstrato de dado é também denominado **tipo opaco**, enquanto um tipo de dado que não promove encapsulamento é classificado como **transparente**.

Uma **estrutura de dados** é uma implementação de um tipo de dado opaco ou transparente. Mais precisamente, uma estrutura de dados mostra como um tipo cujos valores são estruturados pode ser implementado.

Quando um cliente lida com uma estrutura de dados, ele deve se concentrar apenas nas operações oferecidas pela interface da estrutura de dados e não como essas operações são implementadas ou como valores desse tipo são representados em memória.

5.3 Algoritmos

Um **algoritmo** é um conjunto determinado de instruções que, quando seguidas, desempenham uma tarefa particular. Em outras palavras, um algoritmo consiste num conjunto de instruções que, ao serem seguidas, resolvem um determinado problema. Enfim, um algoritmo é um procedimento passo a passo para executar uma tarefa num intervalo de tempo finito.

O primeiro passo para a resolução de um problema por meio de um programa de computador é a definição *precisa* dele. Depois desse passo, planeja-se a solução do problema por meio da escrita de um algoritmo. Um algoritmo consiste numa sequência de passos (instruções) que recebem alguns valores como **entrada** e produzem alguns valores como **saída**. Quando executadas, as instruções de um algoritmo resolvem um determinado problema. Além disso, essas instruções não devem ser ambíguas e a resolução do problema deve encerrar em algum instante.

Uma analogia bastante comum que ajuda no entendimento do conceito de algoritmo é aquela entre algoritmo e receita culinária. Numa receita culinária, os ingredientes e utensílios utilizados (p. ex., ovos, farinha de trigo, assadeira) compõem a entrada e o produto final (p. ex., um bolo) é a saída. O modo de preparo da receita especifica uma sequência de passos que informam como processar a entrada a fim de produzir a saída desejada.

Raramente, um algoritmo é concebido com o objetivo de receber um conjunto limitado de valores. Isto é, mais comumente, um algoritmo é desenvolvido para lidar com vários **casos de entrada**. Por exemplo, um algoritmo criado para resolver equações do segundo grau pode receber como entrada a equação $x^2 - 5x + 6$ e produzir como saída as raízes 2 e 3. Esse mesmo algoritmo serviria para resolver a equação $x^2 - 4x + 4$, produzindo 2 como saída. Nesse exemplo, as equações são casos de entrada do algoritmo exemplificado. Em linguagem cotidiana, um caso de entrada é referido apenas como *entrada*.

Um algoritmo é **correto** quando, para cada caso de entrada, ele encerra após produzir a saída correta. Um algoritmo **incorreto** pode não parar quando ele recebe um determinado caso de entrada ou pode parar apresentando um resultado que não é correto.

Pode haver vários **algoritmos funcionalmente equivalentes** que resolvem um mesmo problema. Algoritmos funcionalmente equivalentes podem usar mais ou menos recursos, ter um número maior ou menor de instruções e assim por diante.

É importante ressaltar ainda que nem todo problema possui algoritmo. Por exemplo, não existe algoritmo para o problema de enriquecimento financeiro (lícito ou ilícito). Esse problema não possui algoritmo porque

sequer é bem definido. Mas, existem problemas que, apesar de serem bem definidos, não possuem algoritmos completos como, por exemplo, jogo de xadrez. É interessante notar ainda que problemas que são resolvidos trivialmente por seres humanos, como falar uma língua natural ou reconhecer um rosto, também não possuem solução algorítmica. Por outro lado, problemas relativamente difíceis para seres humanos, como multiplicar dois números inteiros com mais de dez dígitos, possuem algoritmos relativamente triviais.

Existe uma íntima relação entre construção de algoritmos e estruturas de dados. Ou seja, decisões sobre organização de dados devem ser tomadas com conhecimento dos algoritmos que serão aplicados aos dados e, vice-versa, a escolha de algoritmos depende das estruturas de dados sobre as quais os algoritmos atuam. Assim uma estrutura de dados e um algoritmo devem ser considerados como uma unidade em que nenhum dos dois faz sentido sozinho.

5.3.1 Abordagem de Refinamentos Sucessivos

A abordagem mais comum utilizada na construção de algoritmos é denominada **método de refinamentos sucessivos**. Utilizando essa abordagem, divide-se sucessivamente um problema em subproblemas cada vez menores até que eles possam ser resolvidos trivialmente. As soluções para os subproblemas são então combinadas de modo a resultar na solução para o problema original.

5.3.2 Linguagem Algorítmica (Pseudolinguagem)

Um algoritmo pode ser escrito em qualquer linguagem, como uma língua natural (p. ex., português) ou uma linguagem de programação (p. ex., C). Aliás, um programa de computador consiste exatamente de um algoritmo (ou uma coleção de algoritmos) escrito numa linguagem de programação. Linguagem natural pura raramente é usada na escrita de algoritmos, pois ela apresenta problemas inerentes, tais como prolixidade, imprecisão, ambiguidade e dependência de contexto. O uso de uma linguagem de programação de alto nível também não é conveniente para a escrita de algoritmos complicados, pois o programador precisa dividir sua atenção entre essa tarefa e detalhes sobre construções da linguagem na qual o algoritmo será escrito.

Uma ideia que facilita a vida do programador consiste em usar, na construção de algoritmos, uma linguagem próxima à linguagem natural do programador, mas que incorpore construções semelhantes àquelas encontradas comumente em linguagens de programação. Uma linguagem com essas características é denominada **linguagem algorítmica** ou **pseudolinguagem**. Para atender à finalidade a que se destina, uma linguagem algorítmica precisa ainda ser bem mais fácil de usar do que qualquer linguagem de programação.

Pseudocódigo é um algoritmo escrito numa linguagem algorítmica (pseudolinguagem). Pseudocódigo é usado não apenas por programadores iniciantes, mas também por programadores experientes, embora estes sejam capazes de escrever programas relativamente simples sem o auxílio de pseudocódigo. Deve-se notar que pseudocódigo é dirigido para pessoas, e não para máquinas. Portanto não existe tradutor de pseudocódigo para linguagem de máquina.

5.3.3 Como Construir Algoritmos

Utilizando uma linguagem algorítmica, o desenvolvimento de um programa é dividido em duas grandes fases:

1. **Construção do algoritmo usando linguagem algorítmica**. Nessa fase, o programador deverá estar envolvido essencialmente com o raciocínio que norteia a escrita do algoritmo.
2. **Tradução do algoritmo para uma linguagem de programação**. Aqui, a preocupação do programador não deverá mais ser o raciocínio envolvido na construção do algoritmo. Ou seja, nessa fase, o programador utilizará apenas seu conhecimento sobre uma linguagem de programação para transformar um algoritmo em programa.

A construção de um algoritmo segue, normalmente, a seguinte sequência de passos:

1. **Análise do problema**. Nessa etapa, o enunciado do problema deve ser analisado minuciosamente até que os dados de entrada e saída possam ser devidamente identificados. Feito isso, tenta-se descrever um procedimento em linguagem algorítmica (pseudolinguagem) ou mesmo em língua portuguesa que mostre como obter o resultado desejado (saída) usando os dados disponíveis (entrada). O resultado dessa etapa é um esboço de algoritmo que deverá ser refinado na próxima etapa.
2. **Refinamento do algoritmo**. Nessa etapa, cada passo do algoritmo esboçado na primeira etapa que não tenha solução trivial deve ser subdividido. Isto é, a abordagem de refinamentos deve ser aplicada sucessivamente até que cada subproblema possa ser considerado trivial. O nível de detalhamento de cada instrução resultante depende do grau de conhecimento do programador relativo ao problema e à linguagem de programação para a qual o algoritmo será traduzido. Se o pseudocódigo resultante dessa etapa for difícil de ler ou traduzir numa linguagem de programação, deve haver algo errado com o nível de detalhamento adotado.
3. **Teste do algoritmo**. Após obter um algoritmo refinado, você deve testá-lo. Para tal, verifique se o algoritmo realmente funciona conforme o esperado. Ou melhor, teste o algoritmo com alguns casos de entrada que sejam qualitativamente diferentes e verifique se ele produz a saída desejada para cada um desses casos. Se o algoritmo produz respostas indesejáveis, deve-se retornar a uma das etapas anteriores. Para testar um algoritmo, você deve fazer papel tanto de computador quanto de usuário. Isto é, você deverá executar o algoritmo manualmente, como se fosse um computador, e também deverá fornecer dados de entrada para o algoritmo, como se fosse um usuário.

5.3.4 Paradigmas Algorítmicos

Um **paradigma** é uma abordagem geral de resolução de problemas. Paradigmas existem em diversas áreas de conhecimento. Em Física, por exemplo, existem vários paradigmas: mecânica clássica, mecânica relativista e mecânica quântica que resolvem a mesma categoria de problemas, mas cada um deles é mais adequado para uma situação específica.

Um **paradigma algorítmico** é uma abordagem geral guiada por um conceito particular que pode ser usada na construção de algoritmos. A seguir, serão descritos vários paradigmas algorítmicos, mas existem outros tais paradigmas além daqueles descritos aqui (p.ex., algoritmos de programação dinâmica, algoritmos probabilísticos).

Algoritmos de Divisão e Conquista

Um **algoritmo de divisão e conquista** divide um problema em subproblemas menores e resolve-os independentemente. Um dos algoritmos de divisão e conquista mais simples é a técnica de busca binária, que será explorada na **Seção 7.2.3**. Para resolver um problema, um algoritmo dessa natureza reduz o problema a um número fixo de subproblemas menores do mesmo tipo, cada um dos quais pode ser reduzido a outros subproblemas e assim por diante até que se obtenham subproblemas que podem ser resolvidos trivialmente. Por exemplo, na busca binária o intervalo de busca é continuamente reduzido à metade. Frequentemente (p.ex., busca binária), mas nem sempre (p.ex., torres de Hanói), esses algoritmos são bem eficientes e seus tempos de execução crescem de modo logarítmico (v. **Seção 6.11.4**).

Algoritmos de Retrocesso

A técnica de retrocesso, utilizada na resolução de problemas de satisfação de restrições, foi discutida na **Seção 4.5**. Essa técnica é tipicamente implementada por meio de recursão.

Algoritmos Vorazes

Algoritmos vorazes são usados em problemas de otimização determinados em encontrar uma solução que minimiza ou maximiza algum valor. Essa abordagem sempre considera a melhor escolha em cada passo, em vez de considerar todas as sequências de passos que possam conduzir a uma solução ótima. Um exemplo de algoritmo voraz é a Codificação de Huffman, que será apresentada na **Seção 12.8.2**.

Algoritmos de Força Bruta

Algoritmos de força bruta constituem o paradigma algorítmico mais básico. Um **algoritmo de força bruta** resolve um problema do modo mais direto possível com base no enunciados do problema que ele se propõe a resolver. Tipicamente, um algoritmo de força bruta enumera todos candidatos a solução de um problema e, então, verifica se algum candidato realmente satisfaz à solução do problema. Outra característica da técnica de força bruta é que ela não leva em consideração os recursos computacionais (i.e., tempo de processamento e memória) necessários para resolução de um problema. Enfim, algoritmos de força bruta não consideram possíveis detalhes ocultos no enunciado de um problema. Por exemplo, um algoritmo dessa natureza pode determinar o maior ou menor elemento de um array ordenado de inteiros sem levar em consideração que o array é ordenado, o que simplificaria a solução do problema. Alguns exemplos de algoritmos de força bruta são:

- Ordenação de arrays utilizando o método *Bubble Sort* (v. **Seção 6.11.3**).
- Soma de números inteiros compreendidos entre *1* e *n* usando laço de repetição ou recursão (v. **Seção 4.1**).
- Cálculo de exponenciação como faz a função `ExponenciacaoQuad()`, discutida na **Seção 4.8.5**.

Apesar de serem tipicamente ineficientes, algoritmos de força bruta são frequentemente úteis.

5.4 Tipos de Dados Transparentes e TADs em C

Em C, idealmente, um tipo opaco ou transparente deve ser implementado por intermédio de dois arquivos:

[1] Arquivo de cabeçalho, que provê a interface lógica de acesso às operações disponibilizadas pelo tipo. Idealmente, nesse arquivo aparecem apenas alusões às funções que implementam essas operações e a definição do tipo, que pode ser completa, no caso de tipo transparente, ou incompleta, no caso de tipo opaco (ou TAD — v. **Seção 5.2**).

[2] Arquivo de implementação no qual as funções cujas alusões aparecem no arquivo de interface associado são implementadas.

A **Seção 5.5.1** mostra como a abordagem delineada acima é levada a efeito para um tipo transparente.

Um tipo abstrato de dado (TAD) pode ser implementado em C utilizando-se o procedimento descrito a seguir:

[1] Na interface do tipo (i.e., no arquivo de cabeçalho que contém sua especificação), ele deve ser definido como um ponteiro para um tipo incompleto. Um **tipo incompleto** pode ser um tipo de estrutura ou união cujos membros não tenham sido ainda pormenorizados ou um tipo de array cujo número de elementos não tenha sido especificado ainda. Na prática, tipos incompletos servem para fazer alusões a tipos. Por exemplo, a definição do tipo `tComplexo` a seguir informa o compilador que variáveis desse tipo são ponteiros para estruturas com rótulo `complexo` que serão definidas em outro local do programa.

```
typedef struct complexo *tComplexo;
```

Note que, como mostra o exemplo acima, no caso de definições incompletas de estruturas, o uso do rótulo se faz necessário (v. **Seção 3.10**).

Um ponteiro, como aqueles do tipo `tComplexo`, que aponta para uma estrutura parcialmente definida é denominado **ponteiro opaco**.

[2] No arquivo de implementação do tipo (i.e., aquele com extensão `.c`), o tipo incompleto deve ser completado. Um tipo incompleto torna-se completo quando a informação ausente é especificada dentro do escopo no qual ele é aludido. No caso de estruturas e uniões incompletas, a informação que deve ser especificada são os nomes e os tipos dos campos, enquanto, no caso de arrays incompletos, deve-se especificar o número de elementos. Por exemplo, o tipo de estrutura para a qual os ponteiros do tipo `tComplexo` apontam pode se tornar completa com a seguinte declaração:

```
struct complexo {
            double pReal;
            double pImaginaria;
       };
```

Note que o mesmo rótulo de estrutura utilizado na definição do tipo `tComplexo` é utilizado novamente nessa última declaração.

[3] Variáveis do tipo para o qual o ponteiro aponta devem ser alocadas dinamicamente no arquivo de implementação do tipo. Além disso, deve-se prover uma função para criação e iniciação de variáveis desse tipo, como, por exemplo, a função `CriaComplexo()` a seguir:

```
tComplexo CriaComplexo(double x, double y)
{
   tComplexo umComplexo = malloc(sizeof(struct complexo));

   umComplexo->pReal = x;
   umComplexo->pImaginaria = y;

   return umComplexo;
}
```

A função **malloc()** aloca um espaço em memória com um número de bytes igual ao valor recebido como parâmetro e retorna o endereço desse espaço. Essa função faz parte da biblioteca padrão de C (`#include <stdlib.h>`) e será discutida em detalhes no **Capítulo 9**.

[4] Uma função para liberação do espaço alocado para cada variável do tipo deve ser implementada no referido arquivo de implementação. Essa função pode ser implementada como mostrado a seguir:

```
void DestroiComplexo(tComplexo2 *c)
{
   free(*c);

   *c = NULL;
}
```

A função **free()** libera o espaço alocado dinamicamente por meio de uma chamada de **malloc()** ou qualquer outra função de alocação dinâmica de memória e será explorada no **Capítulo 9**.

A **Tabela 5–1** resume o exemplo explorado nos passos [1] e [2] descritos acima considerando que os arquivos de cabeçalho e de implementação são respectivamente denominados `Complexo.h` e `Complexo.c`.

Arquivo Complexo.h	Arquivo Complexo.c

`typedef struct complexo *tComplexo;`	`#include Complexo.h`
	
	`struct complexo {`
	`  double pReal;`
	`  double pImaginaria;`
	`};`

Tabela 5–1: Exemplo (Parcial) de Definição e Implementação de TAD em C

O problema central que aflige a implementação de tipos abstratos de dados em C é o fato de ser propensa a escoamento de memória. Quer dizer, como os dados são alocados dinamicamente [v. função `CriaComplexo()` acima] deve haver uma maneira de liberá-los explicitamente. Uma solução paliativa para esse problema é oferecer uma operação [como, por exemplo, a função `DestroiComplexo()` apresentada acima] para liberar o espaço ocupado pelos dados alocados dinamicamente. Mas, essa solução não resolve o problema por duas razões: **(1)** não há garantia que o cliente chamará tal função para liberar cada espaço alocado dinamicamente e **(2)** o cliente pode não ter sequer um ponteiro para dados alocados dinamicamente que possa ser passado como parâmetro para tal função. Por exemplo, é possível que o endereço retornado por uma função, como `IniciaComplexo()` acima, seja passado diretamente como parâmetro para outra função, em vez de ser atribuído a um ponteiro (v. exemplo na **Seção 5.5.2**).

Uma solução mais sofisticada, mas difícil de implementar, é aquela usada por algumas linguagens orientadas a objeto e denominada **contagem de referência**. Nessa abordagem, o código gerado pelo compilador é munido de um dispositivo para checar, em intervalos regulares de tempo, se um dado espaço alocado dinamicamente continua sendo referenciado (i.e., possui algum ponteiro apontando para ele). Se nenhuma referência for encontrada para um dado espaço em memória, ele é liberado.

5.5 Exemplos de Programação

5.5.1 Número Complexo como um Tipo de Dado Transparente

Problema: (a) Defina o tipo de dado `tComplexo` que represente números complexos munido das as seguintes operações:

- Criação de um número complexo com protótipo:

```
tComplexo CriaComplexo(double x, double y)
```

- Soma de dois números complexos com protótipo:

```
tComplexo SomaComplexos(tComplexo c1, tComplexo c2)
```

- Apresentação de um número complexo na tela com protótipo:

```
void ExibeComplexo(tComplexo c)
```

(b) Escreva um programa em C para testar o tipo `tComplexo` bem como algumas operações especificadas no item (a).

Solução de (a):

Interface

```
#ifndef _Complexos1_H_
#define _Complexos1_H_

typedef struct {
            double pReal;
            double pImaginaria;
       } tComplexo;

extern tComplexo CriaComplexo(double x, double y);
extern tComplexo SomaComplexos(tComplexo c1, tComplexo c2);
extern void ExibeComplexo(tComplexo c);

#endif
```

Implementação

```
#include <stdio.h>      /* printf() */
#include "Complexos1.h" /* Interface do tipo complexo */

/****
 * CriaComplexo(): Atribui um valor a um número complexo
 *
 * Parâmetros:
 *    x, y (entrada) - partes real e imaginária a serem atribuídas ao número complexo
 *
 * Retorno: O resultado da atribuição
 ****/
tComplexo CriaComplexo(double x, double y)
{
   tComplexo umComplexo;

   umComplexo.pReal = x;
   umComplexo.pImaginaria = y;

   return umComplexo;
}

/****
 * SomaComplexos(): Soma dois números complexos
 *
 * Parâmetros: c1, c2 (entrada) - os dois números que serão somados
 *
 * Retorno: O resultado da soma
 ****/
tComplexo SomaComplexos(tComplexo c1, tComplexo c2)
{
   tComplexo resultado;

   resultado.pReal = c1.pReal + c2.pReal;
   resultado.pImaginaria = c1.pImaginaria + c2.pImaginaria;

   return resultado;
}

/****
 * ExibeComplexo(): Exibe um número complexo na tela
 *
 * Parâmetros: c (entrada) - o número complexo que será exibido
 *
 * Retorno: Nada
 ****/
void ExibeComplexo(tComplexo c)
{
   if (c.pImaginaria < 0)
      printf("%5.2f - %5.2fi", c.pReal, c.pImaginaria);
   else if (c.pImaginaria > 0)
      printf("%5.2f + %5.2fi", c.pReal, c.pImaginaria);
   else
      printf("%5.2f", c.pReal);
}
```

Solução de (b):

```
#include <stdio.h>      /* printf() */
#include "Complexos1.h" /* Interface do tipo complexo */

int main (void)
{
   tComplexo c1 = IniciaComplexo(2, 3),
             c2 = IniciaComplexo(3, 2),
             c3;

   ExibeComplexo(c1);
   printf(" + ");
   ExibeComplexo(c2);
   printf(" = ");

   ExibeComplexo(SomaComplexos(c1, c2));

   c3.pReal = c1.pReal + c2.pReal;
   c3.pImaginaria = c1.pImaginaria + c2.pImaginaria;

   printf("\n\nSoma = %5.2f + %5.2fi\n", c3.pReal, c3.pImaginaria);

   return 0;
}
```

Resultado de execução do programa:

```
2.00 +  3.00i +  3.00 +  2.00i =  5.00 +  5.00i

Soma =  5.00 +  5.00i
```

5.5.2 Número Complexos como um TAD

Problema: (a) Defina o TAD `tComplexo2` que represente números complexos munido das as seguintes operações:

- Criação de um número complexo com protótipo:
  `tComplexo2 CriaComplexo(double x, double y)`
- Destruição de um número complexo (v. **Seção 5.4**) com protótipo:
  `void DestroiComplexo(tComplexo2 *c)`
- Soma de dois números complexos com protótipo:
  `tComplexo2 SomaComplexos(tComplexo2 c1, tComplexo2 c2)`
- Apresentação de um número complexo na tela com protótipo:
  `void ExibeComplexo(tComplexo2 c)`

(b) Escreva um programa em C para testar o tipo `tComplexo` bem como algumas das operações especificadas no item (a).

Solução de (a):

Interface

```
#ifndef _Complexos2_H_
#define _Complexos2_H_

    /* Definição incompleta do tipo tComplexo2 */
typedef struct complexo *tComplexo2;

extern tComplexo2 IniciaComplexo(double x, double y);
extern tComplexo2 SomaComplexos(tComplexo2 c1, tComplexo2 c2);
extern void ExibeComplexo(tComplexo2 c);

#endif
```

Implementação

```
#include <stdio.h>      /* printf() */
#include <stdlib.h>     /* malloc() */
#include "Complexos2.h" /* Interface do tipo tComplexo2 */

struct complexo { /* Completa a definição do tipo tComplexo2 */
             double pReal;
             double pImaginaria;
        };

/****
 * IniciaComplexo(): Atribui um valor a um número complexo
 *
 * Parâmetros:
 *    x, y (entrada) - partes real e imaginária a serem atribuídas ao número complexo
 *
 * Retorno: O resultado da atribuição
 ****/
tComplexo2 IniciaComplexo(double x, double y)
{
   tComplexo2 umComplexo = malloc(sizeof(struct complexo));

   umComplexo->pReal = x;
   umComplexo->pImaginaria = y;
   return umComplexo;
}

/****
 * SomaComplexos(): Soma dois números complexos
 *
 * Parâmetros: c1, c2 (entrada) - os dois números que serão somados
 *
 * Retorno: O resultado da soma
 ****/
tComplexo2 SomaComplexos(tComplexo2 c1, tComplexo2 c2)
{
   tComplexo2 resultado = malloc(sizeof(struct complexo));

   resultado->pReal = c1->pReal + c2->pReal;
   resultado->pImaginaria = c1->pImaginaria + c2->pImaginaria;
   return resultado;
}

/****
 * ExibeComplexo(): Exibe um número complexo na tela
 *
 * Parâmetros: c (entrada) - o número complexo que será exibido
 *
 * Retorno: Nada
 ****/
void ExibeComplexo(tComplexo2 c)
{
   if (c->pImaginaria < 0)
      printf("%5.2f - %5.2fi", c->pReal, c->pImaginaria);
   else if (c->pImaginaria > 0)
      printf("%5.2f + %5.2fi", c->pReal, c->pImaginaria);
   else
      printf("%5.2f", c->pReal);
}
```

Solução de (b):

```
#include <stdio.h>      /* printf() */
#include "Complexos2.h" /* Interface do tipo tComplexo2 */

/****
 * main(): Testa operações sobre números complexos
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main (void)
{
   tComplexo2 c1 = IniciaComplexo(2, 3),
              c2 = IniciaComplexo(3, 2),
              c3;

   ExibeComplexo(c1);
   printf(" + ");
   ExibeComplexo(c2);
   printf(" = ");

   ExibeComplexo(SomaComplexos(c1, c2));

      /* Se qualquer delimitador de comentário na linhas    */
      /* a seguir for removido, ocorrerá erro de compilação */
//   c3.pReal = c1.pReal + c2.pReal;
//   c3->pReal = c1->pReal + c2->pReal;

   return 0;
}
```

Resultado de execução do programa:

```
2.00 +  3.00i +  3.00 +  2.00i =  5.00 +  5.00i
```

5.6 Exercícios de Revisão

Definições Fundamentais (Seção 5.1)

1. O que se estuda na disciplina Estruturas de Dados?
2. Qual é a importância da organização de dados na construção de um programa?
3. (a) O que é um tipo de dado? (b) O que é um tipo de dado primitivo? (c) O que é um tipo de dado derivado?
4. O que é uma definição de variável?
5. (a) O que é um tipo de dado estruturado? (b) Como são classificados os tipos de dados estruturados?
6. (a) O que é uma variável estruturada? (b) O que é uma variável estruturada homogênea? (c) O que é uma variável estruturada heterogênea?
7. Defina tipo de dado escalar e tipo de dado estruturado.
8. Que informações são providas para o compilador por uma definição de variável?
9. O que é um cliente de um tipo de dado?
10. O que é encapsulamento de dados?
11. O que é ocultação de informação?
12. Qual é a relação entre ocultação de informação e encapsulamento de dados?

Tipos Abstratos e Estruturas de Dados (Seção 5.2)

13. Qual é diferença entre abstração e implementação?

14. (a) O que é uma abstração de alto nível? (b) O que é uma abstração de baixo nível?

15. O que é um tipo abstrato de dado (TAD)?

16. O que é uma estrutura de dados?

17. Uma estrutura de dados é o mesmo que um tipo de dado estruturado?

18. (a) O que é um tipo de dado opaco? (b) E um tipo de dado transparente? Apresente exemplos de cada um deles.

19. (a) Qualquer tipo de dado é um tipo abstrato de dado? (b) Qualquer tipo abstrato de dado é um tipo de dado?

20. (a) Qual é a importância de ocultação de implementação em programação? (b) Por que é aconselhável ocultarem-se detalhes de implementação de um tipo abstrato de dado?

21. De acordo com as definições apresentadas neste capítulo, o tipo **int** de C:

(a) É uma estrutura de dados?

(b) É um tipo de dado?

(c) É um tipo abstrato de dado?

(d) É um tipo opaco ou transparente?

Algoritmos (Seção 5.3)

22. O que é um algoritmo?

23. Qual é a diferença entre algoritmo e programa ou subprograma?

24. (a) Em que aspectos o conceito de algoritmo é análogo ao conceito de receita culinária? (b) Quando essa analogia deixa de ser válida?

25. O que é um caso de entrada?

26. Qual é a relação entre organização de dados e algoritmos?

27. O que são algoritmos funcionalmente equivalentes?

28. Cite dois exemplos de problemas que não possuem solução algorítmica.

29. Descreva a abordagem de refinamentos sucessivos utilizada na construção de algoritmos.

30. (a) O que é linguagem algorítmica? (b) Por que linguagem algorítmica também é denominada *pseudolinguagem*?

31. Qual é a vantagem do uso de pseudolinguagem na construção de algoritmos em detrimento ao uso de uma linguagem natural (p. ex., português)?

32. (a) O que é pseudocódigo? (b) Existe tradutor (i.e., compilador ou interpretador), para algoritmos escritos em pseudocódigo?

33. Quais são as etapas envolvidas na escrita de um algoritmo?

34. Como um problema cuja solução algorítmica é desejada deve ser analisado?

35. Como um algoritmo deve ser testado?

36. (a) O que são casos de entrada qualitativamente diferentes? (b) Qual é a importância do uso de casos de entrada qualitativamente diferentes em testes de algoritmos?

37. Como exemplos de execução de um algoritmo auxiliam em seu processo de desenvolvimento?

38. Por que, mesmo que um algoritmo tenha sido testado e considerado correto, é necessário testar um programa derivado dele?

39. (a) O que é um paradigma? (b) O que é um paradigma algorítmico?

40. Descreva os seguintes paradigmas algorítmicos:

(a) Algoritmo de divisão e conquista
(b) Algoritmo de retrocesso
(c) Algoritmo voraz
(d) Algoritmo de força bruta

Tipos de Dados Transparentes e TADs em C (Seção 5.4)

41. Como um tipo de dado deve ser implementado em C?

42. Quais são as partes constituintes de uma implementação de um tipo abstrato de dado em C?

43. O que é um ponteiro opaco?

44. Qual é o procedimento utilizado na implementação de TADs em C?

45. (a) Qual é o problema que aflige a implementação de tipos abstratos de dados em C? (b) Por que disponibilizar uma função para liberar memória alocada dinamicamente não resolve esse problema?

46. O que é contagem de referência?

Exemplos de Programação (Seção 5.5)

47. Suponha que as variáveis `c1`, `c2` e `c3` sejam do tipo `tComplexo` definido na **Seção 5.5.1**. O que há de errado com a seguinte instrução?

```
c3.pReal = c1.pReal + c2.pReal;
```

48. Suponha que as variáveis `c1`, `c2` e `c3` sejam do tipo `tComplexo2` definido na **Seção 5.5.2**. Por que a instrução a seguir causa erro de compilação?

```
c3.pReal = c1.pReal + c2.pReal;
```

5.7 Exercícios de Programação

EP5.1 Complete o tipo abstrato de dado `tComplexo2` apresentado na **Seção 5.5.2** incluindo as seguintes operações:

(a) Conjugado de um número complexo com protótipo:

```
tComplexo2 ConjugadoComplexo(tComplexo2 c)
```

(b) Módulo de um número complexo com protótipo:

```
double ModuloComplexo(tComplexo2 c)
```

(a) Destruição de um número complexo (i.e., liberação do espaço ocupado por suas partes real e imaginária) com protótipo:

```
void DestroiComplexo(tComplexo2 c)
```

(b) Subtração de dois números complexos com protótipo:

```
tComplexo2 SubtraiComplexos(tComplexo2 c1, tComplexo2 c2)
```

(c) Multiplicação de dois números complexos com protótipo:

```
tComplexo2 MultiplicaComplexos(tComplexo2 c1, tComplexo2 c2)
```

EP5.2 (a) Implemente o tipo de dado `tComplexo` apresentado na **Seção 5.5.1** usando coordenadas polares. (b) Use a mesma função **main()** apresentada na **Seção 5.5.1** e verifique que o novo programa não funciona mais. [**Sugestões:** 1. Se você desconhece coordenadas polares, consulte um texto sobre números complexos. 2. Use as funções **sqrt()** e **atan2()** declaradas no cabeçalho `<math.h>`.]

EP5.3 (a) Implemente o TAD `tComplexo2` apresentado na **Seção 5.5.2** usando coordenadas polares. (b) Use a mesma função **main()** apresentada na **Seção 5.5.2** e verifique se o novo programa funciona tão bem como antes. [**Sugestões:** Siga as mesmas sugestões apresentadas para o exercício **EP5.2**.]

EP5.4 Um número racional é um número que pode ser escrito como a/b, em que a e b são números inteiros, sendo $a \neq 0$. Escreva um tipo abstrato de dado para representar números racionais munido das seguintes operações:

(a) Soma
(b) Multiplicação
(c) Subtração
(d) Divisão

EP5.5 **Preâmbulo:** No espaço tridimensional, vetores são representados por três valores reais nas direções ortogonais que definem o espaço. Assim vetores tridimensionais podem ser representados pelo seguinte tipo:

```
typedef struct {
           double x;
           double y;
           double z;
        } tVetor;
```

As seguintes operações sobre vetores tridimensionais são definidas:

(a) **Multiplicação de um vetor por uma constante**. Se um vetor é representado por `(x, y, z)` e `c` é uma constante, a multiplicação do vetor pela constante resulta no vetor `(cx, cy, cz)`.

(b) **Soma de dois vetores**. Se dois vetores são representados por `(x1, y1, z1)` e `(x2, y2, z2)`, a soma deles é o vetor `(x1 + x2, y1 + y2, z1 + z2)`.

(c) **Norma de um vetor**. Se o vetor é representado por `(x, y, z)`, sua norma é a raiz quadrada positiva de $x^2 + y^2 + z^2$.

(d) **Produto interno de dois vetores**. O produto interno dos vetores representados por `(x1, y1, z1)` e `(x2, y2, z2)` é dado por `x1*y1 + x2*y2 + z1*z2`.

(e) **Produto vetorial de dois vetores**. O produto vetorial dos vetores representados por `(x1, y1, z1)` e `(x2, y2, z2)` é o vetor `(y1*z2 - z1*y2, z1*x2 - x1*z2, x1*y2 - y1*x2)`.

Problema: (a) Escreva um tipo abstrato de dado para representar vetores no espaço tridimensional munido das operações descritas acima. (b) Implemente um programa que seja cliente desse tipo e teste suas operações.

EP5.6 (a) Escreva uma função que recebe um array de elementos do tipo `tComplexo` definido na **Seção 5.5.1**, calcula a soma dos elementos do array e retorna o endereço do resultado, que também é armazenado no terceiro parâmetro da função, cujo protótipo é:

```
tComplexo *SomaArrayComplexos( const tComplexo ar[], int nElementos,
                               tComplexo *soma )
```

(b) Escreva uma função, denominada `LeComplexo()`, que lê um valor do tipo introduzido via teclado. (c) Escreva um programa que lê um array de elementos do tipo `tComplexo` usando a função `LeComplexo()`, soma os elementos do referido array usando a função solicitada no item (a) e apresenta o resultado na tela utilizando a função `ExibeComplexo()` apresentada na **Seção 5.5.1**.

EP5.7 Considere o tipo `tPonto`, que representa pontos no plano euclidiano, definido como:

```
typedef struct {
           double x, y;
        } tPonto;
```

(a) Escreva uma função que lê um ponto do plano cartesiano e armazena-o numa estrutura do tipo `tPonto`. O parâmetro único dessa função deve ser um ponteiro para essa estrutura.

(b) Escreva uma função que recebe dois parâmetros do tipo `tPonto` que representam pontos do plano cartesiano e apresenta na tela a equação da reta que passa pelos dois pontos.

(c) Escreva uma função que recebe dois parâmetros do tipo `tPonto` e retorna a distância entre esses pontos.

(d) Escreva uma função que recebe três parâmetros do tipo `tPonto`, que representam pontos do plano cartesiano e retorna `1` se eles forem colineares ou `0`, em caso contrário.

(e) Escreva um programa que lê valores para duas variáveis do tipo `tPonto` usando a função solicitada no item (a) e chama a função solicitada no item (b) para escrever a equação da reta que passa pelos dois pontos representados por essas variáveis.

[**Sugestão:** Consulte um bom texto sobre Geometria Analítica elementar para saber como se determina a equação da reta que passa por dois pontos, como se calcula a distância entre dois pontos no plano cartesiano e para saber como se verifica se três pontos do plano cartesiano são colineares.]

EP5.8 Suponha que intervalos de tempo sejam representados num programa em C por estruturas do tipo definido como:

```
typedef struct {
           int hora;
           int min;
        } tTempo;
```

(a) Escreva uma função que calcula a soma de dois intervalos de tempo.

(b) Escreva uma função que subtrai dois intervalos de tempo.

(c) Escreva uma função que lê um intervalo de tempo.

(d) Escreva uma função que exibe na tela um intervalo de tempo.

(e) Escreva um programa que testa as funções propostas nos itens de (a) a (d).

EP5.9 Escreva um TAD que implemente operações sobre o tipo `tData` que representam datas e é definido como:

```
typedef struct {
           int dia, mes, ano;
        } tData;
```

As operações a ser disponibilizadas por esse TAD são as seguintes:

(i) Leitura e validação de datas (v. **Seção 3.12.3**)

(ii) Exibição de datas na tela

(iii) Cálculo do tempo decorrido entre duas datas

EP5.10 Transforme o tipo `tVetor`, definido na **Seção 3.12.4** e que representa vetores no espaço euclidiano bidimensional, num TAD munido das operações definidas naquela seção.

EP5.11 Transforme o tipo `tPonto`, definido na **Seção 3.12.5** e que representa pontos em coordenadas retangulares e polares, num TAD munido das operações definidas naquela seção.