Princípios de Programação com Exemplos em C

Aleardo Manacero Jr.

Notas de Aula - 2022

Dedico este texto aos meus filhos (Gustavo e Miguel) e minha esposa (Ciça), por permitirem minhas horas adicionais de trabalho.

Prefácio

Quando esse texto começou a ser escrito o objetivo era simplesmente o de ter um material de boa qualidade que apresentasse os conceitos introdutórios sobre programação na ordem que considerava adequada. Com o início da empreitada foi possível perceber que escrever um texto introdutório sobre programação de computadores é uma tarefa extremamente complexa, como mostrarei a seguir. Mesmo assim a qualidade do material, após muitas revisões e correções, melhorou a ponto de torná-lo publicável e assim o objetivo passou de minhas próprias aulas para o apoio ao ensino de programação segundo uma abordagem baseada em algoritmos e linguagens imperativas. É esse o texto que o leitor encontra a seguir. Antes, porém, é preciso apontar alguns aspectos relevantes sobre o texto, seu uso e sua aplicabilidade.

Primeiro, temos que entender que escrever um texto introdutório sobre programação de computadores (e provavelmente sobre vários outros assuntos) é algo muito complexo. O principal problema na elaboração do texto é explicar cada assunto num nível de detalhamento que permita ao leitor compreender corretamente aquilo que está escrito. A dificuldade nisso é que os tópicos tratados neste texto são, há muito tempo, parte integrante da forma de pensar de quem atua no ensino de programação. Isto resulta em que grande parte dos assuntos tratados parece ser óbvia quando na realidade não o são (pelo menos não tão óbvias quanto parecem). Assim, em alguns momentos as explicações parecem ser exageradas e desnecessárias. De fato em alguns momentos elas realmente são desnecessárias e nesse ponto reside a difícil decisão de identificar o que realmente é necessário e o que apenas gerará tédio ao leitor (como talvez esse parágrafo esteja fazendo).

Outro problema a ser resolvido é definir o foco do texto. Esse problema, felizmente, depende mais das preferências do autor, embora possa levar leitores incautos a considerarem que faltam tópicos ao texto. O foco do texto define, em última análise, o uso do mesmo e sua aplicabilidade. Assim dedico os próximos parágrafos ao exame deste assunto.

O foco deste texto

A arte ou ciência de programar máquinas já existe desde o início do século XIX, quando foram desenvolvidos os primeiros teares programáveis. Nesses mais de duzentos anos a atividade de programação deixou os cartões dos teares para ir aos modernos computadores digitais, em que os programas são constituídos por informações armazenadas em meios magnéticos e podem fazer com que bilhões de operações sejam realizadas num piscar de olhos. Durante essa evolução ela passou de uma arte para uma ciência, em que os comandos a serem executados pelos computadores podem ser modelados e planejados com precisão e confiabilidade.

É exatamente essa mudança de arte para ciência que determina o foco deste texto. Nosso entendimento é de que a aprendizagem de programação deve ser feita através de métodos formalmente definidos, em que o uso de uma ou outra linguagem de programação não faça diferença alguma. Assim, o conteúdo aqui apresentado parte do princípio de que a essência da programação está na ciência da modelagem do problema e definição dos algoritmos. Assim, este é o foco do texto e não a implementação de programas em uma linguagem qualquer.

A definição do foco sobre aspectos conceituais de programação e não sobre aspectos técnicos da mesma faz com que diversos aspectos relativos à linguagem fiquem de fora do texto. O que se quer aqui é fornecer um texto que permita ao leitor escrever programas a partir de uma boa definição de conceitos de estruturas de dados e de modularização do problema. A inclusão de exemplos em linguagem C (junto com os conceitos básicos dessa linguagem) ocorreu apenas para tornar o texto menos entediante. Não se pretende apresentar aqui um curso de linguagem C (muito menos fazer uso nos exemplos de técnicas mais elaboradas de codificação, que aproveitem recursos não usuais da linguagem). Para esse fim existem bons livros e os excelentes manuais da linguagem, que são perfeitamente acessíveis para quem já compreendeu os mecanismos da boa programação.

A distribuição de conteúdos

Os conceitos de programação de computadores apresentados nesse texto aparecem numa sequência lógica de complexidade e aplicabilidade. O texto inicia com uma rápida introdução sobre os conceitos fundamentais de computação, mostrando como o computador funciona e como programas devem ser desenvolvidos. Nesse sentido são brevemente descritos alguns conceitos de hardware e de aspectos relativos a algoritmos, modelos para solução de problemas e programação estruturada. Nesse capítulo inicial apresenta-se também os conceitos preliminares de linguagem C.

Após essa introdução se faz uma pequena incursão ao estudo de tipos de dados, com foco nos tipos mais fundamentais de uma linguagem de programação (tipos numéricos simples e tipos caracteres).

Na sequência do texto se faz um exame cuidadoso de estruturas de controle de fluxo em programas. No capítulo 3 examinam-se os testes de decisão, deixando os laços de repetição para o capítulo 4. Deve-se deixar claro, entretanto, que o exame dessas estruturas de controle (testes e repetição) é precedida por um breve estudo dos conceitos sobre testes de condição, operadores condicionais e operadores lógicos, uma vez que esses são a base para as estruturas de controle.

Tendo o conhecimento sobre as estruturas de controle é possível passar-se ao uso de estruturas de dados mais elaboradas. Assim, o capítulo 5 trata da apresentação de estruturas homogêneas de dados (os vetores). Nesse ponto também é apresentado o uso de endereços como um tipo simples de dados (mais precisamente o uso de ponteiros para endereços), uma vez que os mesmos serão necessários para determinadas passagens de parâmetros para funções nos exemplos em linguagem C do capítulo seguinte.

Nesse ponto do texto já é possível resolver problemas mais complexos, o que torna necessário modularizar nossos programas. Assim no capítulo 6 são apresentados os subprogramas. São examinados com cuidado os conceitos sobre escopo de variáveis e passagem de parâmetros. O uso de recursão e de programas recursivos é deixado para um capítulo posterior. O leitor pode, entretanto, adiantar esses conceitos se considerar mais conveniente. Na sequência aqui imaginada o mecanismo de recursão apenas se torna obrigatório na manipulação de listas, servindo então como sua introdução.

O capítulo anterior praticamente encerra a apresentação das técnicas de programação disponíveis. No restante do texto o que se faz é apresentar progressivamente novas estruturas de dados. Assim, no capítulo 7 são apresentadas as estruturas de dados heterogêneas e, aproveitando a sua aplicabilidade, a manipulação básica de arquivos de dados.

O que falta fazer para este texto se tornar um livro

No momento o material atende de modo bastante satisfatório os dois primeiros semestres de um curso de computação. Falta-lhe ainda material tratando de estruturas de dados não lineares, como as árvores binárias, incluindo formas de se fazer o balanceamento dessas árvores. Faltam ainda alguns algoritmos mais especializados, como filas de prioridade, métodos de ordenação não comparativos, etc. Já as estruturas de dados para armazenamento secundário (árvores B, entre outras) não serão tratadas, mesmo na versão completa do texto.

Outro aspecto a ser melhorado no texto diz respeito aos exemplos e listas de exercícios. Hoje eles aparecem numa quantidade mínima necessária e precisam ser ampliados ainda mais para melhorar seu aspecto didático.

Tudo isso deve ser feito em breve, com a colaboração dos alunos que experimentam esse material ano a ano. Por isso meu muito obrigado a todos vocês.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Este texto foi escrito com o objetivo de criar o primeiro contato entre um futuro programador de computadores e os computadores e as linguagens que permitem sua programação, considerando inicialmente o paradigma imperativo de programação e, consequentemente, uso de uma linguagem imperativa. Com isso em mente, esse primeiro capítulo faz uma breve introdução ao que se entende por programação estruturada de computadores. Isso inclui a apresentação de alguns aspectos da máquina (o computador) para que se possa ter uma compreensão mais adequada de como os programas são executados. Inclui também uma introdução ao método estruturado de programação, passando por sua modelagem, construção dos algoritmos e respectiva implementação.

Essa introdução aos conceitos de programação estruturada deve ser vista com muito cuidado. Isso porque apesar de ser uma metodologia aparentemente natural, muitos acabam tendo dificuldades na atividade de programação por não seguirem seus conceitos de forma sistemática. A escolha por usar-se uma estratégia baseada no paradigma imperativo é que quando seu aprendizado é sólido, fica muito mais simples mover-se para outros paradigmas de programação, em especial, quando se usam ambientes de programação modernos, em que o trabalho de projeto muitas vezes é negligenciado.

Nos próximos capítulos são apresentados aspectos particulares da programação de computadores. Esses aspectos surgem dentro de uma sequência em que a complexidade dos programas e problemas resolvidos aumenta progressivamente. Assim, no capítulo 3 são apresentadas as principais estruturas que permitem controlar o fluxo de execução de um programa (isto é, a ordem em que os comandos presentes serão executados). Essas estruturas de controle são essenciais na implementação de qualquer programa, uma vez que mesmo problemas simples envolvem a realização de processos repetitivos (uma ação executada várias vezes) ou de escolha por caminhos alternativos (a partir de um critério de decisão qualquer).

Uma vez apresentadas as principais estruturas de controle passamos ao exame dos tipos de valores que podem ser tratados por um computador no capítulo 2. Apesar de na prática o computador entender apenas os sinais ligado/desligado, podemos criar estruturas que representam valores inteiros, reais e caracteres, entre outros, na forma de sinais ligado/desligado. Na prática, o uso do computador apenas faz sentido através da manipulação de valores desses tipos, organizados tanto de forma individual como coletiva.

Conhecendo então as estruturas de controle e os tipos de dados fundamentais temos que ao longo do capítulo 6 apresenta-se um conceito essencial para que se resolva problemas mais complexos. Esse conceito é o de dividir o problema em partes menores, mais fáceis de se resolver. Essa técnica resulta na criação de subprogramas, que existem em várias formas nas diversas linguagens de programação.

Finalmente voltamos, ao longo do capítulo 7, ao exame de como representar valores no computador. Agora, entretanto, nos preocupamos com a representação de valores mais complexos, que incluam a composição de valores distintos como nomes e números através de um único nome simbólico. Isso inclui o uso de tais valores em locais de armazenamento externos à memória, que chamamos arquivos.

Ao final desses capítulos espera-se que o leitor tenha condições de escrever programas relativamente complexos. Programas escritos a partir desse material falham apenas por não executarem necessariamente de modo eficiente. Essa eficiência surge através do exame de métodos para busca e ordenação de informações, que são tipicamente parte de cursos de estrutura de dados.

Vamos então iniciar nosso estudo de programação através do exame de como o computador funciona e de quais são os aspectos principais de programação estruturada. Antes de prosseguir devo deixar claro que as siglas usadas ao longo deste texto seguem um padrão que o autor considera mais comumente usado, com alguns termos usados no seu original em inglês e outros traduzidos, quando a tradução tiver uso amplo na literatura técnica brasileira.

1.1 O computador

Um computador nada mais é do que uma máquina que quando devidamente preparada, através de seus programas, pode realizar tarefas relativamente complexas. As tarefas que podem ser cumpridas são sempre manipulações numéricas e/ou simbólicas sobre conjuntos de dados, isto é, manipulações sobre informações. Foi nesse contexto que ocorreu, na realidade, a origem do termo *informática*.

Para realizar estas manipulações um computador necessita de certos componentes, alguns para se comunicar com o mundo externo (chamados dispositivos de entrada e/ou saída), outros para armazenar dados e programas (as memórias)

e ainda uma unidade de processamento (a**CPU**, de *Central Processing Unit*). Na figura 1.1 está apresentada a organização destes componentes para que o computador funcione, segundo a chamada "arquitetura de Von Neumann".

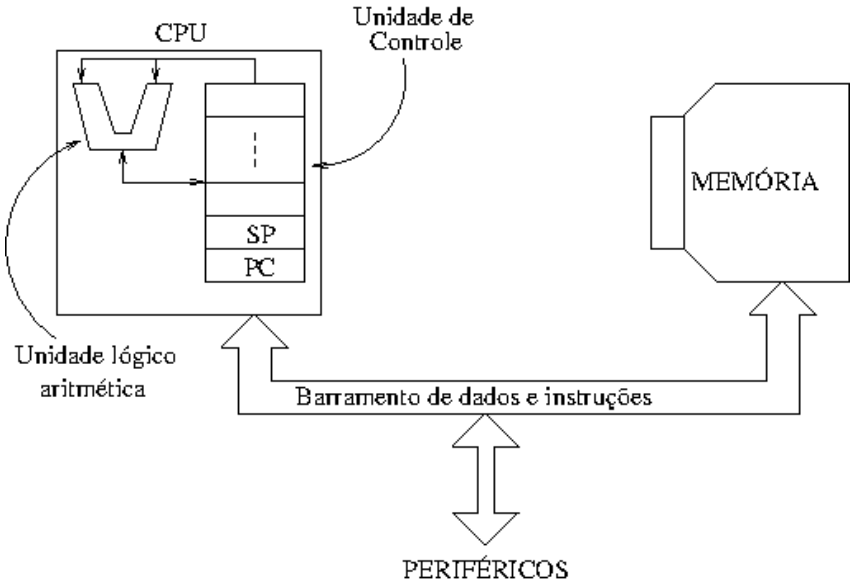

Figura 1.1: Componentes básicos de um computador.

A execução de um programa ocorre de forma bastante simples. Os procedimentos apresentados a seguir representam corretamente, embora de forma simplificada, como ocorre a execução de um programa no computador. Deve-se ressaltar que esses procedimentos são praticamente os mesmos desde os primeiros computadores até hoje, exceto por aspectos técnicos que permitem um melhor aproveitamento dos ciclos de relógio da máquina.

1. Inicialmente as instruções para a execução de um programa ficam em algum dispositivo de armazenamento externo, como um disco rígido (o HD), ou CD-ROM, ou outro dispositivo qualquer. Para que o programa possa ser executado é preciso que ele seja levado para a memória do computador, numa operação definida como carregamento do programa, realizada pelo carregador ou *loader*.

2. Uma vez carregados os programas podem ser executados pelo sistema operacional (que é quem controla todo o funcionamento do computador). Isso ocorre através da passagem, sequencialmente, das instruções que compõem o programa para a CPU, onde elas serão efetivamente executadas na unidade lógico-aritmética (ULA).

3. Para a execução na ULA a instrução é previamente decodificada na unidade de controle (UC), quando se determina qual a ação a ser executada e quais os dados necessários para a sua execução. Os dados necessários são trazidos então da memória ou lidos a partir de algum periférico, como teclado, disco rígido ou qualquer outro que permita a entrada de dados.

4. Uma vez concluída a execução de uma instrução a UC define qual a próxima instrução a ser executada, a partir do resultado obtido e do valor do chamado contador de programas (PC). Essa instrução é então trazida para a CPU e o processo se repete até que se chegue ao final do programa.

Durante a execução do programa, sempre que necessário, instruções específicas dirigem o computador a transferir informações armazenadas na memória para dispositivos periféricos, tais como arquivos e monitores, entre outros dispositivos de saída de dados. Podem também trabalhar em sentido contrário, isto é, solicitar que dispositivos de entrada de dados (arquivos, teclado, *mouse*) passem informações para a memória, onde ficariam armazenadas e disponíveis para uso na CPU.

As ações descritas aqui representam os passos básicos para a execução de programas em computadores digitais. O entendimento dessas ações permite que se compreenda a razão para que os vários comandos e tipos de dados existentes em uma linguagem resultem nos objetivos pretendidos pelo programador. Podemos agora passar para o exame de como são os programas e as linguagens de programação.

1.2 Programas e linguagens de programação

Como dito anteriormente, um computador é apenas uma máquina que precisa ser preparada para funcionar. Esta preparação nada mais é do que um conjunto de instruções mostrando o que deve ser feito a cada instante. Esse conjunto de instruções é chamado "programa".

Entretanto, como o computador é uma máquina, as instruções devem seguir um formato compreensível por ele. Este formato consiste em sequências de sinais elétricos que representam cada instrução usando os estados "ligado" e "desligado" representando valores lógicos (verdadeiro ou falso) como padrão de referência. Assim, o trecho de programa da figura 1.2 representa as instruções para colocar o valor zero em algumas posições de memória de computadores equipados com o processador 8085 (usado nos primeiros PCs).

```
00000110010100001010111101110111001000110000010 1
11000010000001100000000011000011000011000000000 0
```

Figura 1.2: Programa para zerar posições de memória (processador 8085)

Cabe aqui abrirmos um parenteses para definir alguns nomes usados para os uns e zeros que aparecem nessa figura. Cada dígito desses recebe o nome de **bit**, que vem de **b**inary dig**it**, sendo que o computador trabalha com conjuntos de bits chamados **palavras**. Um tamanho especial de palavra é o de oito bits, que é a unidade convencional de medida de armazenamento conhecida por **byte**. As instruções apresentadas na figura 1.2 ocupam portanto 12 bytes, ou 96 bits.

Claro que este monte de 'uns' e 'zeros' é completamente ilegível para um ser humano (pelo menos para os normais). O primeiro passo para contornar este problema foi a criação de linguagens de montagem (*assembly*), com as quais o mesmo programa seria escrito na forma da figura 1.3.

```
        lxi h, ender
        mvi b, comp
zera:   xra a
        mov m, a
        inx h
        dcr b
        jnz zera
```

Figura 1.3: Programa para zerar posições de memória (assembly para 8085).

Esse programa, entretanto, continua a parecer um hieróglifo para quem não tem o conhecimento sobre o funcionamento do processador (CPU) específico. Além disso, cada tipo (ou família) de processador possui um conjunto de instruções diferente, que torna praticamente impossível transportar um programa escrito em *assembly* de um tipo de CPU para um computador que use um outro tipo de CPU.

O ideal seria que fosse possível escrever programas em nossa linguagem que fossem compreensíveis pelo computador. Isso ainda é impossível pela natureza intrinsecamente complexa da gramática presente nas chamadas linguagens naturais (português, inglês, etc.). A solução encontrada é o uso de linguagens inter-

mediárias, com uma gramática e vocabulário bem mais restrito, que permitam a escrita de um programa com relativa facilidade e que possam ser facilmente traduzidas para a linguagem de máquina específica.

O processo de tradução é feito com o uso de programas especiais, que convertem um programa escrito numa linguagem pré-definida (o programa ou código fonte) na linguagem da máquina (código executável). Estes programas especiais são de dois tipos: os **interpretadores**, nos quais o programa é convertido no momento em que é executado; e os **compiladores**, nos quais é feita uma conversão prévia do fonte para a linguagem de máquina, sendo que o resultado dessa conversão é armazenado num arquivo chamado de binário, ou executável, que poderá ser ativado (executado) quantas vezes for preciso.

As linguagens nas quais serão escritos os programas são mais próximas de expressões matemáticas perfeitamente compreensíveis para seres humanos. Dentre as muitas linguagens de programação existentes podemos citar pascal, fortran, C, Java, lisp, prolog, python, cobol, etc. A seguir, na figura 1.4, aparece escrito em pascal o programa para zerar 50 posições de memória, propositadamente com erros de sintaxe (a declaração de ARRAY) e de lógica (a comparação feita no comando WHILE).

```
PROGRAM zeramem;

VAR
   INTEGER  mem;
   mempos: ARRAY [1..50] of INTEGER;

BEGIN
   mem := 1;
   WHILE (mem < 50) DO
   BEGIN
      mempos[mem] := 0;
      mem := mem + 1;
   END;
END.
```

Figura 1.4: Programa para zerar posições de memória (pascal).

Nem todas as linguagens funcionam da mesma maneira, uma vez que problemas diferentes exigem técnicas de solução também diferentes. Assim, as muitas linguagens de programação podem ser classificadas segundo vários critérios. Um

deles é o nível de abstração da linguagem, em que temos linguagens de alto nível como o pascal e linguagens de baixo nível, como o C. O nível de abstração é a medida da proximidade entre a linguagem do programa e da máquina, sendo que as de alto nível estão mais distantes da máquina.

Uma outra maneira de classificar linguagens é segundo seu grau de estruturação. Uma linguagem estruturada é aquela que possui mecanismos (comandos da linguagem) capazes de organizar suas ações através de padrões que podem ser logicamente formalizados. O pascal e o C são exemplos de linguagens estruturadas, enquanto o fortran (em sua forma original) não é estruturado.

Finalmente, uma última forma de classificar uma linguagem é através de seu paradigma de programação, isto é, como programas em uma determinada linguagem devem ser "pensados". Nessa classificação temos essencialmente os paradigmas:

- imperativo (pascal, C, fortran, entre muitas), em que os programas são compostos por listas de comandos ou ordens de como fazer as as coisas, que são executadas de forma linear, adequando-se fortemente ao conceito da arquitetura de Von Neumann;
- funcional (Lisp, scheme, haskell, ML), em que os programas são compostos por funções (aplicações) que traduzem o que deve ser feito, em geral através de uma formulação matemática aplicada aos elementos de um determinado conjunto;
- lógico (prolog), em que os programas são compostos por séries de proposições lógicas, associadas a regras de predicados em lógica de primeira ordem, sendo que sua execução não é linear, dependendo de quais regras são verdadeiras a cada momento da busca pela solução;
- orientado a objetos (smalltalk, C++, Java, C#, python), em que os programas executam ações, ou métodos, sobre objetos que representam eventos e elementos do mundo real, sendo que as linguagens desse paradigma são originadas em grande parte de linguagens imperativas.

Alguns autores ainda acrescentam os paradigmas concorrente (Ada) e modular (Modula-2), mas esses podem ser entendidos como especializações, para aplicações específicas, dos paradigmas anteriores.

1.3 Algoritmos e programação estruturada

Até agora, falamos de programas apenas de forma bastante genérica, sem nos preocupar em como um programa é escrito. Pois bem, antes de termos um programa

nas mãos é necessário que executemos alguns passos iniciais, indo do problema a ser resolvido até o programa escrito numa linguagem de computação qualquer. A seguir temos a descrição desses passos.

Definição do problema

Se existe algum problema a ser resolvido temos que defini-lo de forma bastante clara antes de fazer qualquer outra coisa. Esse processo faz parte do que se chama engenharia de requisitos, que é a atividade de determinar com precisão o que o usuário deseja e como isso pode ser fornecido pelo programa. A arte de definir corretamente o problema nos economiza muito tempo de desenvolvimento, pois citando C.F. Kettering, "Um problema bem definido é um problema metade resolvido".

Apesar de ser uma etapa aparentemente irrelevante, a definição do problema não é tão simples quanto parece. Ao longo de muitos anos de prática pude observar que a maioria dos problemas enfrentados no desenvolvimento de projetos é o entendimento do que precisa ser feito. Casos simples, como os que aparecem nos exemplos desse texto, de fato não representam um obstáculo à definição do problema e de sua solução. Entretanto, o aumento da complexidade do problema faz com que seja mais difícil defini-lo de modo correto e, consequentemente, encontrar o modelo que o resolva.

Apenas para exemplificar, considere que um astrônomo deseja calcular quanto tempo levaria para três planetas, hoje com suas órbitas alinhadas em relação ao sol, terem novamente suas órbitas alinhadas. Para resolver corretamente este problema falta definir a que alinhamento ele se refere, isto é, se é na mesma posição atual ou não. São questões como essa que devem ser completamente resolvidas antes de se dar qualquer outro passo na criação de um programa computacional.

Geração de um modelo

O segundo passo no processo é criar um modelo que defina de forma única o problema em questão. Modelos bem construídos facilitam o projeto do programa, uma vez que permitem restringir nosso trabalho ao que é realmente necessário e ainda nos guiam na geração de soluções. Em linhas gerais podemos pensar no modelo de um programa como sendo equivalente à maquete de um prédio, que permite identificar como ele ficará (e corrigir eventuais erros de projeto) a um custo muito pequeno e num tempo bem menor do que sua real construção.

De modo geral, bons modelos usam equações matemáticas para representar o problema a ser resolvido. O uso de modelos matemáticos se explica pois é mais

fácil transformar equações em instruções de uma linguagem de programação do que fazer o mesmo com enunciados abstratos em formulações puramente textuais. O formalismo matemático é usado mesmo em situações aparentemente fora desse contexto, pois sempre é possível transformar um problema em uma representação simbólica que permita algum grau de equacionamento matemático.

Voltando ao exemplo do astrônomo, é na definição do modelo que aparece toda a diferença entre o alinhamento ser na mesma posição ou não. No primeiro caso o modelo é muito simples, bastando que se calcule o mínimo múltiplo comum entre os períodos das órbitas dos três planetas. No segundo caso o alinhamento ocorrerá muito antes, devendo ser calculado a partir um sistema de equações para as posições angulares dos três planetas, sendo que o alinhamento ocorrerá no primeiro instante em que todos tenham o mesmo momento angular.

Criação de um algoritmo

Um algoritmo nada mais é do que uma receita para a transformação do modelo num conjunto de ações que possam ser reescritas em linguagem de programação. Idealmente, um algoritmo em sua versão final é quase o programa escrito em linguagem natural. O processo de criação de um algoritmo segue uma abordagem de refinamentos sucessivos (chamada de ***top-down***), em que se parte de um algoritmo muito rudimentar, tal como *"para ir até a biblioteca vá até a biblioteca"*, do qual refinam-se as instruções até que todas sejam facilmente executáveis, como *"para ir até a biblioteca siga pela rua X até o cruzamento com a rua Y, vire a direita na rua Y e siga por dois quarteirões, até o número Z, que é onde fica a biblioteca"*.

Voltando mais uma vez ao problema do nosso astrônomo, o algoritmo teria que descrever, passo a passo, como resolver todas as equações e cálculos para o respectivo modelo. Para o caso simples, teríamos um algoritmo de alto nível dizendo, por exemplo: encontre os períodos das órbitas dos três planetas numa unidade pré-definida (dias, anos), calcule o m.m.c desses períodos, que será o tempo necessário, na unidade pré-definida, para se ter o novo alinhamento na mesma posição.

O algoritmo detalhado teria que explicar como se obtém os períodos na tal unidade, depois como se calcula o m.m.c de três números a partir de sua decomposição em primos (explicando como se faz essa decomposição) e depois definir ainda como o resultado final será apresentado.

Implementação do algoritmo

A partir do algoritmo passamos a escrever o programa na linguagem de programação desejada. A escolha entre as várias linguagens disponíveis deve ser feita segundo critérios como eficiência, familiaridade e disponibilidade da mesma. A etapa de implementação é extremamente mecânica, sendo que existem até programas que geram código fonte numa determinada linguagem a partir de uma especificação formal[1].

Apenas para confirmar isto, dentro da engenharia de software se considera que esta etapa é a que deveria consumir a menor parte do tempo de projeto e, de fato, se torna a mais curta se as demais forem bem feitas.

Como se pode ver, a criação de um programa não é uma tarefa complexa se seguirmos uma estrutura organizada de passos **(um algoritmo !!)**, a partir do problema até o programa.

Na implementação de sistemas mais complexos existem técnicas específicas de modelagem, originadas a partir da engenharia de software, tais como os modelos baseados em diagramas de fluxo (DFD) e, num nível mais alto o padrão UML por exemplo. Esses modelos, além de criarem uma sistemática para o desenvolvimento de softwares, permitem a associação gráfica para que ações o programa irá executar.

Numa escala mais simples de modelagem temos os fluxogramas, que podem ser usados para problemas menos complexos. Um fluxograma é, na realidade, uma alternativa também ao algoritmo do programa, uma vez que também pode descrever todos os passos do algoritmo de forma gráfica. Um fluxograma utiliza sinais gráficos específicos ("blocos") para representar classes de ações a serem tomadas em cada passo. Na figura 1.5 são apresentados os principais blocos usados em fluxogramas. Esses blocos incluem o de processo, que representa a execução de comandos (explicitados no bloco) para alguma ação; o de decisão, que representa comandos para a escolha entre dois ou mais caminhos; o de entrada de dados e o de saída de dados (impressão, gravação em arquivos), entre outros.

O processo completo de geração do programa depende muito pouco da linguagem a ser utilizada na implementação. Da definição do problema ao algoritmo o que mais importa é o paradigma a ser usado (lembrem-se que paradigma pode ser entendido como a forma de pensar o programa). Apenas na etapa de imple-

[1]Especificação formal é uma técnica de se especificar as ações de um programa a partir de critérios algebricamente consolidados. Isso equivale, de certo modo, à escrita de um algoritmo usando uma linguagem baseada em relações algébricas e, permite grandes ganhos na construção de programas funcionalmente corretos.

mentação é que surge a dependência da linguagem. Assim, se temos um algoritmo (ou fluxograma) bem detalhado, fica bastante simples escrever-se um programa que resolva nosso problema em quase qualquer linguagem (claro que tomando-se o cuidado com eventuais restrições oriundas do paradigma utilizado).

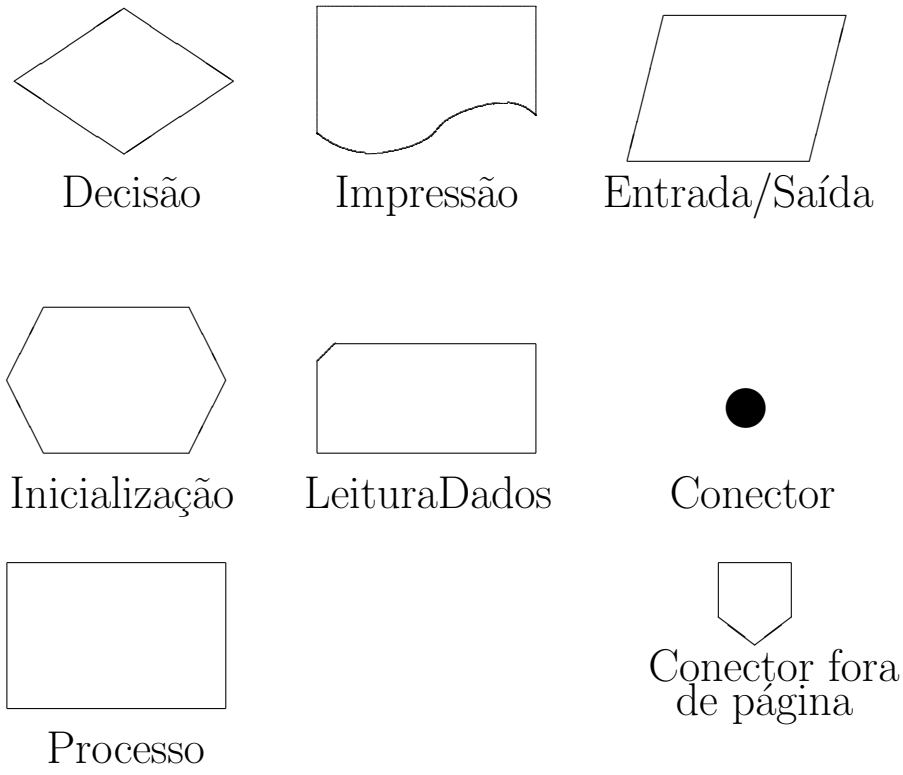

Figura 1.5: Blocos usados em fluxogramas.

Exemplo

Considere que lhe solicitem um programa que calcule o fatorial de um número natural qualquer. As etapas de geração desse programa são as seguintes:

1. Definição do problema

   Nesse caso essa etapa é bastante 'natural'. Nosso problema é apresentar um programa que calcule o fatorial de um número natural, lembrando que os naturais são formados pelos inteiros maiores ou iguais a zero.

2. Geração do modelo

   O cálculo do fatorial pode ser representado pelas seguintes equações matemáticas:

   $$n! = 1, \text{ se } n=0 \text{ ou } n=1$$

   ou

   $$n! = n \times (n-1) \times \ldots \times 3 \times 2 \times 1, \text{ se } n > 1$$

3. Criação do algoritmo

   Considerando a equação apresentada no modelo, um possível algoritmo aparece na Figura 1.6, mais adiante. Para chegar até aquele algoritmo pode-se pensar inicialmente num algoritmo base como sendo a equação dada pelo modelo. A partir dela podemos imaginar que antes de calcular o fatorial temos que saber qual o número desejado (óbvio!!), portanto temos que ler esse número, acusando um erro caso não seja um natural. Depois temos que saber se ele é igual a zero ou um. Se for a resposta é imediata, ou seja, fatorial será igual a 1. Se não for, temos que usar a segunda parte da equação, ou seja, calcular o produto indicado. Para fazer isso podemos repetir a operação de multiplicação k vezes (sendo k igual ao número menos 1), da forma como aparece na equação.

```
LEIA um número
FAÇA fatorial igual ao número
SE número menor que zero
   ENTÃO escreva que não existe fatorial de inteiros negativos
SE número igual a zero ou número igual a um
   ENTÃO
      FAÇA fatorial igual a um
   SENÃO
      DECREMENTE o número
      ENQUANTO número for maior que zero
         REPITA DAQUI:
            FAÇA fatorial igual a fatorial vezes o número
            DECREMENTE o número
      ATÉ AQUI
ESCREVA o valor de fatorial
```

Figura 1.6: Algoritmo para cálculo de fatorial.

4. Implementação do algoritmo

   Com o algoritmo detalhado da Figura 1.6 é possível gerar o código fonte para o programa em qualquer linguagem do paradigma imperativo (e para muitas outras linguagens de outros paradigmas). Em nosso exemplo usaremos a linguagem C.

```
#include <stdio.h>

main( )
{int num, fatorial;

    puts ("Digite um número");
    scanf ("%d", &num);   // faz a leitura do número
    if (num < 0)  // testa se e negativo, se for encerra
        { puts("Não existe fatorial de número negativo");
          exit(0);
        }

    fatorial = num;
    if ((num == 0) || (num == 1))  // testa para n=0 e n=1
        fatorial = 1;
      else // se for diferente calcula o fatorial pela regra
        { num = num - 1;
          while (num > 0) // repete as multiplicações para
                          // cálculo do fatorial
          { fatorial = fatorial * num;
            num = num - 1;
          }
        }
    // apresenta o resultado do fatorial
    printf (" O fatorial é %d \n", fatorial);
}
```

Figura 1.7: Programa na linguagem C para cálculo de fatorial.

1.3.1 Programação estruturada

Uma vez definidos os passos a serem executados para a criação de um programa, temos que nos preocupar com o estilo a ser utilizado no processo. A *programação estruturada* é o caminho correto a ser seguido em termos de estilo, pois assegura ao programador a clareza necessária para a construção de programas corretos e eficientes.

Para se escrever um programa de forma estruturada é necessário seguir algumas regras fundamentais que dizem respeito ao fluxo de execução do programa. Estas regras podem ser melhor entendidas de forma gráfica, usando os blocos básicos de fluxogramas para indicar os fluxos de execução a serem seguidos pelo processador durante o atendimento das regras. Temos então:

1. Regra da modularização:

   Deve-se **modularizar** o programa. Isso significa particionar um problema em sub-problemas menores e de solução mais simples, como feito na criação de um algoritmo. Este processo muitas vezes recebe o nome de "dividir para conquistar" em outras áreas de trabalho. O diagrama da figura 1.8 apresenta claramente esse conceito.

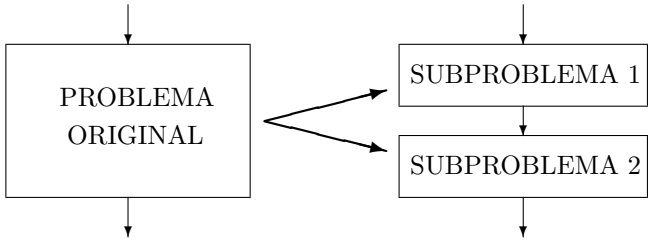

Figura 1.8: Modularização de tarefas.

2. Regra da unicidade de caminhos

   Cada módulo **deve ter precisamente uma entrada e uma saída**. Isso implica, em termos gerais, que todo programa deve ter um início e um final. Essa regra é útil para determinar como um programa se comporta durante sua execução, uma vez que em qualquer momento teremos sempre um único caminho a seguir a partir de um dado módulo.

   Isso aparentemente não é verdade para o bloco de decisão em um fluxograma, em que a partir dele partem dois caminhos (um quando o resultado do teste é verdadeiro e outro quando é falso). Nesse caso temos que o módulo por ele definido deve conter necessariamente os dois ramos que saem do bloco, de forma a garantir que haverá apenas uma saída, como visto na 1.9. Esta estratégia é conveniente pois assim teríamos formas bastante simples de controlar o fluxo do programa, podendo prever com exatidão quais seriam os caminhos a serem percorridos e também se esses serão aqueles que realmente deveriam ter sido percorridos.

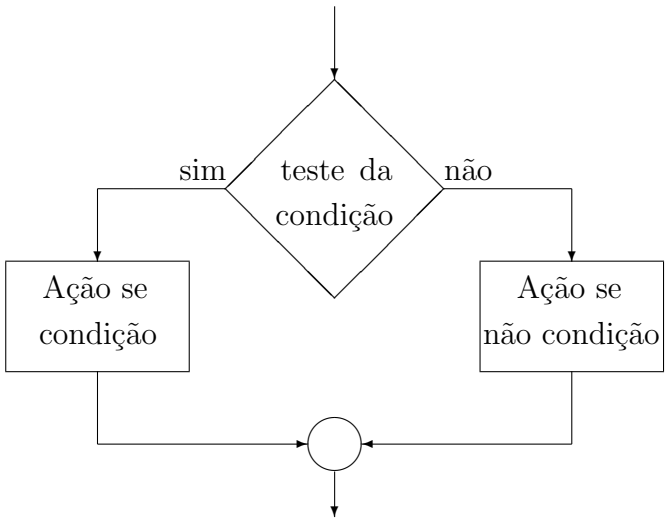

Figura 1.9: Unicidade de acesso e saída em módulos de decisão.

Essa prática se repete com os módulos de repetição, que são aqueles usados para controlar atividades repetitivas, como as multiplicações do exemplo de fatorial. Na prática as repetições são construídas a partir de um teste de decisão (devo ou não continuar a repetir) e um corpo de atividades (aquilo

que estou repetindo).

No caso de blocos de decisão existe ainda uma diferenciação quanto ao momento em que o teste de decisão pela primeira vez. Isso resulta em dois tipos distintos de blocos de decisão: aqueles em que o corpo é executado pelo menos uma vez, e aqueles em que isso pode não ocorrer. A Figura 1.10 apresenta os dois tipos de módulos de repetição e como os módulos são construídos para atender a regra de unicidade de acesso e saída.

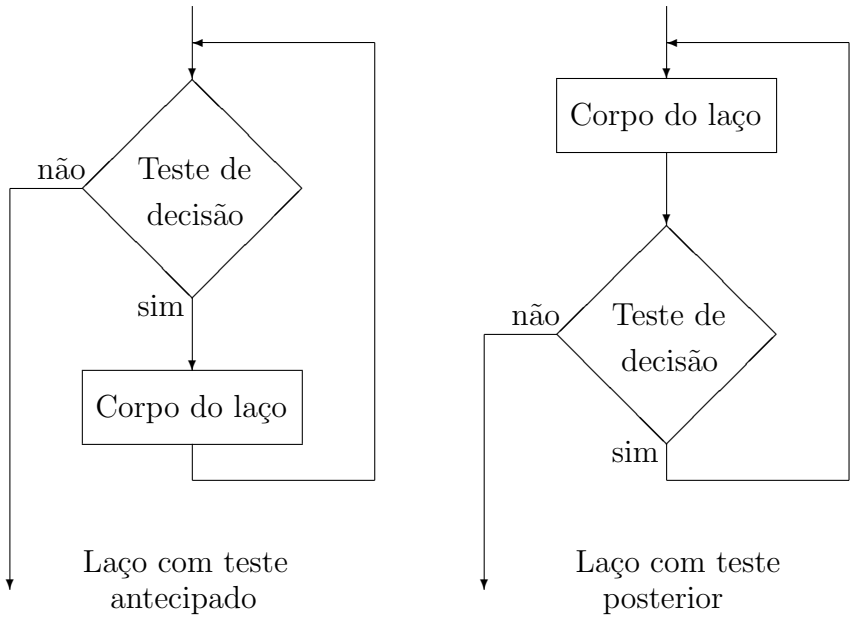

Figura 1.10: Unicidade de acesso e saída em módulos de repetição.

As chances de obter um programa funcionalmente correto aumentam substancialmente quando um programador utiliza essas regras básicas no momento de especificar o algoritmo que resolve seu problema. Além disso, como programas seguindo essa filosofia acabam por ser bastante legíveis, a manutenção (ato de acrescentar ou modificar tarefas executadas pelo programa) e a obtenção de programas corretos ficam muito mais simples. Essa facilidade aumenta ainda mais se o programador tomar o cuidado de colocar comentários significativos, acrescentar linhas em branco separando módulos e colocar espaços de tabulação (chamados de indentação) para visualizar o grau de aninhamento entre módulos, como ocorre com o exemplo da figura 1.7.

Como uma nota adicional, existem métodos exatos para verificar se um pro-

grama está correto. O uso dos chamados métodos formais é baseado em lógica de primeira ordem e álgebra, sendo essenciais em áreas de computação em que o rigor das respostas fornecidas pelo computador seja elevado (com possíveis prejuízos de vida ou materiais se assim não forem).

1.4 A linguagem C e alguns aspectos de legibilidade de código

Embora já seja uma linguagem relativamente antiga, do início dos anos 70, a linguagem C ainda é intensamente utilizada. Do ponto de vista de aprendizagem C tem suas vantagens e desvantagens. Primeiro se trata de uma linguagem com gramática base bastante simples, seguindo muito a estrutura de um algoritmo. Uma segunda vantagem em aprender C como a primeira linguagem reside no fato de já se aprender uma linguagem de uso profissional, o que não ocorre por exemplo com Pascal, que é mais didática porém tem pouco uso nos dias de hoje. Do lado das desvantagens temos o fato de ser largamente dependente do uso de bibliotecas para que o programa funcione e de não fazer previamente a verificação de conflitos e exceções, o que dificulta a identificação de problemas de consistência do programa.

Como o essencial num curso introdutório é a apresentação do estilo correto de programação, ou seja, estruturação de programas, construção de algoritmos e modelagem de problemas, nos preocuparemos apenas com as construções mais elementares da linguagem C, deixando para depois o uso de comandos e estruturas mais complexas e poderosas. Entre as estruturas que serão inicialmente deixadas de lado estão os chamados ponteiros para endereços de memória e seu uso em estruturas dinâmicas. Procuraremos, ao longo dos próximos capítulos, nos limitar a apenas aquilo que for necessário para compreender e usar eficientemente as ideias de modularização e unicidade de caminhos.

Agora faremos uma breve excursão até as características mais básicas do C, as quais podem ser transportadas (com as devidas correções gramaticais) a qualquer outra linguagem de programação estruturada.

1.4.1 Uso de nomes simbólicos

Nome simbólico (ou nome de variável ou simplesmente variável) é o nome que o programador usa para identificar posições de memória que contenham determinada informação. Por exemplo, no programa que calcula o fatorial, nós usamos o nome simbólico “fatorial” para representar a posição física, na memória do computador, em que estava armazenado o valor do fatorial durante o seu cálculo.

Numa analogia bastante simplista, poderíamos dizer que a memória seria um escaninho e que cada posição do escaninho é identificada externamente por um nome afixado nela (esse seria o nome simbólico da posição). Assim, o programa pode ser visto como sendo um conjunto de operações, tais como leitura de valores, comparações e escrita de valores, sobre os conteúdos deste escaninho.

Para que a associação entre nomes e espaços da memória funcione corretamente é preciso, antes, definir como as posições de memória serão alocadas, ou seja, quantos bytes são necessários para armazenar cada tipo de informação. Diferentes tipos de variáveis ocuparão quantidades de memória também distintas, ou seja, se a variável estiver representando uma letra (ou caractere), o espaço ocupado será menor do que se o valor representado fosse um número inteiro ou real.

C admite vários tipos básicos de variáveis, representando caracteres, números inteiros e reais com diferentes precisões. Assim, quando precisamos reservar espaço para um valor inteiro devemos declarar isso explicitamente no programa, através de um comando de declaração de variáveis. Os tipos mais básicos em C são *int* (representando valores inteiros), *float* (reais), *double* (reais com precisão dobrada) e *char* (caracteres). Além desses tipos básicos as variáveis numéricas apresentam modificadores de tamanho (*long* e *short*) ou de sinal (*unsigned*). No capítulo 2 voltaremos a tratar com mais detalhes os tipos básicos da linguagem.

A declaração de um nome simbólico para um dado qualquer segue uma sintaxe bastante simples, consistindo simplesmente em algo do tipo:

TIPO_DA_VARIÁVEL nome_da_variável ;

ou

TIPO_DA_VARIÁVEL nome_da_variável_1, nome_da_variável_2, ... ;

Observe-se que toda declaração deve terminar com um ponto-e-vírgula, enquanto listas de variáveis de mesmo tipo têm os nomes dessas variáveis separados por vírgulas.

Apesar de aceitar infinitos nomes diferentes, C (como qualquer linguagem de programação) apresenta algumas limitações nos nomes possíveis. Assim, temos três regras a serem obedecidas na formação de um nome de variável, que são:

1. Um nome pode conter combinações de letras, números e o caractere '_' (*underline* ou sublinhado), sem espaços em branco entre cada caractere. Vale observar que existe distinção entre letras maiúsculas e minúsculas.

2. Um nome deve sempre começar com uma letra, que pode ser seguida por qualquer dos caracteres da regra 1.

3. Um nome não pode ser uma das palavras reservadas da linguagem, tais

como *if, else, while, do, for, char, int, float, double, long, short, switch, case* e *default*, entre outras. Essa lista depende do compilador C utilizado e deve, portanto, ser verificada nos manuais para os casos menos usuais.

Ao atender essas regras as declarações a seguir são válidas em C:

```
int var1, nome_complicado, v14432, c;
float var2, era_uma_vez, a_;
```

Enquanto que os nomes a seguir não são válidos:

```
int 4var, zu*mm;
double nome.errado, _a;
```

1.4.2 Atribuição de valor

Atribuir um valor a uma variável nada mais é do que dizer que a posição de memória ocupada por aquela variável conterá aquele valor. Por exemplo, no programa para cálculo de fatorial apresentado na figura 1.7, atribuímos o valor do número lido à variável **fatorial** antes de iniciarmos o seu cálculo. Toda linguagem de programação possui uma sintaxe particular para realizar operações de atribuição. No caso do C a sintaxe é a seguinte:

VAR_QUE_RECEBE = VALOR_ATRIBUÍDO;

Em que VAR_QUE_RECEBE é sempre uma variável e VALOR_ATRIBUÍDO pode ser uma constante (valor imediato), o conteúdo de uma outra variável ou ainda o resultado da avaliação de uma expressão. Por exemplo, as atribuições a seguir são válidas:

```
VAR1 = 47;
VAR2 = VAR1;
VAR3 = VAR1 + VAR2 * VAR1;
```

Enquanto estas não são válidas

```
2 = 2;               \\ VAR_QUE_RECEBE é uma constante
VAR1 + VAR2 = VAR3;  \\ VAR_QUE_RECEBE é uma expressao
```

1.4.3 Definição de constantes

Um tipo especial de nome simbólico é o referente a posições de memória que terão valor fixo durante a execução do programa, isto é, que não podem ser alterados pelo programa depois de definidos. Estes nomes são declarados como "constantes", e tem valores atribuídos no momento de sua definição. A sintaxe da definição de uma constante em C pode ser vista a seguir:

```
#define a 0
#define b 20.00
#define pi 3.14159265
```

1.4.4 Uso de bibliotecas

Uma das principais características da linguagem C é de que o programador deve dizer explicitamente tudo o que deve ser usado dentro de um programa. Assim, ele deve fazer uso de comandos **macro** que incluam bibliotecas de funções que serão usadas pelo programa. Isso é feito através de chamadas do tipo:

```
#include "nome_da_biblioteca"
#include <nome_da_biblioteca>
```

A diferença entre o uso de "..." ou <...> está na definição de quais diretórios (ou pastas) são examinados na busca pelo arquivo correspondente a 'nome_da_biblioteca'. No caso de "..." o caminho de busca parte do diretório atual, passando pelos diretórios especificados na linha de comando do compilador, terminando com os diretórios padrões do sistema. Já no caso de <...>, o caminho não passa pelo diretório atual, começando direto naqueles especificados ao compilador.

Em geral os nomes das bibliotecas seguem o padrão *nome.h*, quando são arquivos com as definições das estruturas de dados e nomes de funções, ou, excepcionalmente, *nome.c*, quando se tratarem de arquivos contendo código C de fato. Alguns exemplos de bibliotecas padrões relevantes em C incluem:

stdio.h	$\longrightarrow$	deve estar presente em qualquer programa, contendo as definições básicas de entrada e saída de dados
math.h	$\longrightarrow$	contém funções matemáticas não elementares, como as trigonométricas, exponenciais, logarítmicas, etc.
time.h	$\longrightarrow$	funções para manipulação de tempo

1.4.5 Lendo e escrevendo informações

Um programa é inútil se não produzir nem consumir dados. Logo, toda linguagem deve prover mecanismos com os quais o programa possa interagir com o ambiente. Isso é feito por operações de leitura de dados, quando queremos passar uma informação para ser consumida pelo programa, e operações de saída de dados, que ocorrem quando o programa está produzindo alguma informação que possa ser utilizada fora do programa.

Tanto a entrada como a saída de dados podem ser feitas de formas distintas, de acordo com a fonte ou o destino da informação. Um programa pode ler dados a partir de arquivos, do teclado ou mesmo de dispositivos especiais de aquisição de dados. Da mesma forma, ele pode produzir dados para uma impressora, um arquivo ou mesmo para dispositivos de acionamento de outros sistemas ligados ao computador. O tratamento de cada um destes dispositivos é feito de forma bastante semelhante, inclusive com comandos muito semelhantes, tomando-se apenas o cuidado de direcionar (indicar qual) o dispositivo de entrada/saída que será usado.

No caso do C, a leitura de dados é feita através de vários comandos diferentes, dos quais se destaca o comando *scanf*. Já a escrita também pode ser feita por outros tantos comandos, com destaque para os comandos *printf* e *puts*. A sintaxe destes comandos é dada por:

```
printf ( "descrição da formatação de saída", dados de saída ) ;
puts ( "descrição da formatação de saída") ;
scanf ( "descrição da formatação de entrada", variáveis de entrada ) ;
```

A descrição da formatação, no caso de saída, é uma composição entre comandos da linguagem e textos que queremos ver impressos. Na descrição de entrada são comandos que descrevem a formatação dos dados que serão lidos pelo programa. Alguns dos comandos que podem ser usados para essa descrição incluem:

%d	indica a leitura ou escrita de um valor inteiro
%i	indica a leitura ou escrita de um valor inteiro
%ld	indica a leitura ou escrita de um valor inteiro longo
%f	leitura ou escrita de um valor real (*float*)
%lf	leitura ou escrita de um valor real de precisão dupla (*double)*
%c	leitura ou escrita de um símbolo caractere (*char*)
%s	leitura ou escrita de uma cadeia de caracteres (*string*)

Outros caracteres de formatação (tabulação, linha, etc.) ainda podem ser

usados, como:

\n	indica que se deve saltar para a linha seguinte
\t	indica que se deve fazer uma tabulação
\v	indica que se deve fazer tabulação vertical
\b	indica que se deve retroceder um caractere
\r	indica que se deve retroceder ao início da linha

Os dados de saída são compostos por nomes de variáveis ou expressões que terão seus valores (ou resultados) apresentados na saída definida pelos comandos *printf* ou *puts*. Esses nomes, quando mais de um valor será apresentado, são separados por vírgulas. Já as variáveis de entrada são nomes de variáveis, também separadas por vírgulas se mais de uma, que receberão os valores a serem lidos pelo programa. A ordem de aparição das variáveis e expressões (quando for o caso) deve seguir exatamente a ordem definida na descrição de formatação.

Exemplos de comandos de entrada/saída

```
scanf ("%d %s", &alfa, letra); // lê um inteiro e uma palavra
puts ("Digite um inteiro e uma palavra"); // escreve essa frase
printf ("%f = massa e %lf = velocidade\n", massa, velInicial);
   // escreve um float e um double dentro da frase especificada
printf ("%d", alfa); // escreve um inteiro
```

Uma nota adicional deve ser feita ao comando *gets()*. Embora bastante poderoso no que diz respeito ao que consegue fazer, esse comando **deve** ser evitado como alternativa para entrada de dados. Seu modo de operação permite a criação de brechas muito perigosas no acesso e manipulação das informações na memória. Isso faz com que diversos compiladores emitam uma mensagem de aviso de perigo quando encontram o comando *gets()* em algum programa.

1.4.6 Operadores aritméticos básicos

Embora a entrada e saída de dados sejam operações importantes na execução de um programa, o processamento ocorre, de fato, através de comandos de manipulação dos dados lidos. No caso de valores numéricos existem operadores básicos e funções complexas (que serão tratadas apenas no capítulo 2). Os operadores

básicos são:

+	soma de dois operandos
–	subtração de dois operandos ou "negativação" de um operando multiplicação de dois operandos
/	divisão de dois operandos, resultando num valor real para operandos reais ou num valor inteiro (o quociente da divisão inteira) para operandos inteiros
++	incremento (em uma unidade) de um operando
– –	decremento (em uma unidade) de um operando

Em expressões que envolvam vários operadores aritméticos deve-se seguir a ordem de precedência definida matematicamente, podendo, é claro, fazer-se uso de parenteses para forçar precedências diferentes da norma.

1.4.7 Delimitadores e comentários

A clareza e legibilidade de um programa estão fundamentalmente ligadas a dois parâmetros. O primeiro deles é o forte uso de programação estruturada em sua implementação, o que resulta nas vantagens já descritas na seção 1.3.1. O segundo parâmetro que facilita a legibilidade é o uso de comentários sobre o que cada trecho do programa realiza. Assim, um programa bem escrito deve estar sempre bem comentado, para que qualquer outro programador o possa ler, entendendo com rapidez qual sua função e como executa tal tarefa.

Entretanto, as informações contidas nos comentários não podem ser compiladas e executadas pelo computador. Isso ocorre pelo simples fato de que as mesmas estão no programa apenas para que nós, humanos, possamos entender o que está sendo feito, não fazendo sentido algum para o computador. Logo, devemos deixar esses comentários de lado no momento de fazer a compilação, o que é obtido introduzindo-se certos caracteres que indiquem ao compilador que determinados trechos não devem ser compilados. Estes são os caracteres de delimitação de comentários, que existem em todas linguagens de programação. No caso do C os caracteres usados são:

`// comentário` , usado em comentários de apenas uma linha, e
`/* comentário */` , usado em comentários que ocupem uma ou mais linhas.

Além de comentários existem outros delimitadores em C que são bastante

importantes, tais como:

;	$\longrightarrow$	indica o final de um comando ou expressão
,	$\longrightarrow$	separa elementos de uma lista
( )	$\longrightarrow$	servem para compor expressões ou funções
{ }	$\longrightarrow$	indicam o início e fim de um conjunto de comandos sequenciais, que tenham mesmo nível hierárquico

1.4.8 Funções e a função *main*

Um programa em C é construído a partir de um conjunto de funções, que ao serem executadas permitem a realização de alguma tarefa. Todo programa em C deve ter uma função denominada *'main'*, que o compilador reconhece como sendo o ponto de partida do programa, ou seja, o ponto em que o computador começará a sua execução quando for requisitado.

A sintaxe de uma função, inclusive da função *main* segue o seguinte padrão:

```
TipoDaFunção    nomeDaFuncao   ( listadeparametros )
{ //        indica o início da função
      declarações de variáveis locais
      comandos da função
} //        indica o final da função
```

Detalharemos a sintaxe de funções mais adiante, no capítulo sobre subprogramas. Assim, nesse momento apenas damos mais atenção à função *main*, que tem a seguinte sintaxe:

```
main   (int argc, char *argv[] )
{  Declarações de variáveis da função main
            Código principal do programa
}
```

Em que *argc* é uma variável de tipo inteiro que diz quantos parâmetros foram passados ao programa em sua chamada, enquanto *argv* é a lista efetiva desses parâmetros, em que o primeiro membro da lista é o próprio nome do programa. Os parâmetros da função *main* são opcionais, podendo ou não aparecer num programa.

1.4.9 Um *(outro)* programa completo em C

Apenas para reforçar os comandos apresentados nas últimas páginas apresentamos a seguir, na figura 1.11, um outro exemplo de um programa escrito na

linguagem C.

```
/* Programa que ao ler um inteiro retorna o próprio
 número ou ele diminuído de uma unidade */

#include "stdio.h"

main ( )

{int num;

  puts (" Escreva um número menor que 3");
  scanf ("%d", &num);  // lê um inteiro e guarda na variável num
  if ( num == 1)  // se num for 1 executa o próximo comando
       printf (" Você escolheu %d \n", num);
     else   // senão executa esse comando
       printf (" Você não escolheu %d \n", num - 1);
}
```

Figura 1.11: Estrutura básica de um programa em C

EXERCÍCIOS:

1. Quais os dois aspectos fundamentais em programação estruturada?
2. Qual a importância do uso de comentários em um programa?
3. Porque usar indentação em seus programas?
4. O que você entende como clareza em um programa?
5. Que aspectos você considera importantes no momento de se definir qual o problema que será resolvido através de um computador?
6. Descreva, com o máximo de detalhes possível, como ficariam os modelos para os seguintes problemas:
   (a) Identificar palavras iguais a um certo valor em uma lista de palavras
   (b) Identificar se existem duas palavras iguais em uma lista de palavras
   (c) Ordenar alfabeticamente uma lista de palavras
   (d) Calcular o determinante de uma matriz $3x3$
   (e) Determinar quantos metros de arame são necessários, no mínimo, para cercar um pomar em que as posições das árvores são dados por coordenadas geográficas
   (f) Criar um jogador computadorizado para o jogo da velha
7. Escreva algoritmos para os modelos definidos no exercício anterior.

Capítulo 2

Tipos de dados fundamentais

Como transpareceu do capítulo anterior, é muito importante a identificação dos tipos de conteúdos (valores) tratados pelos programas. Neste capítulo examinaremos inicialmente as condições para definição de tipos e variáveis do ponto de vista da resolução computacional de um problema qualquer. Depois serão apresentados os detalhes de como isso é feito em C.

2.1 Tipos de dados

No primeiro capítulo indicou-se que uma variável é, na realidade, um nome simbólico atrelado a um espaço na memória, que fica reservado para armazenar um determinado valor. Na maioria das linguagens exige-se que os nomes simbólicos sejam pré-declarados para que possam ser utilizados. Isso significa, na prática, que o computador precisa saber antecipadamente quanto de memória ele deverá reservar para armazenar o valor representado por um dado nome simbólico[1]. Neste texto nos ateremos ao caso mais geral, exigindo que as variáveis sejam declaradas antes de seu primeiro uso, como é o caso da linguagem C.

A definição de que tipo de valor pode ser armazenado em cada espaço de memória ocorre, na maioria das linguagens, através de declarações feitas antes do uso da variável. O número de bytes necessário em cada caso depende de padrões definidos pelo hardware da máquina. O grau de liberdade sobre quais faixas de valores são admitidas depende ainda de qual a linguagem é usada, embora algumas faixas sejam relativamente padronizadas.

A figura 2.1 mostra a alocação de espaço para duas variáveis (*num* e *maior*), supostamente declaradas como valores numéricos do tipo inteiro. O computador,

[1]Devemos observar que algumas linguagens permitem a ausência de declarações, como Python ou LISP, e outras permitem a não declaração caso o nome simbólico esteja restrito a um certo padrão, como o Fortran.

ao iniciar a execução desse programa reserva 4 bytes (a partir de um certo endereço na memória) para a variável *num* e, na sequência, reserva mais 4 bytes (consecutivos aos primeiros) para a variável *maior*.

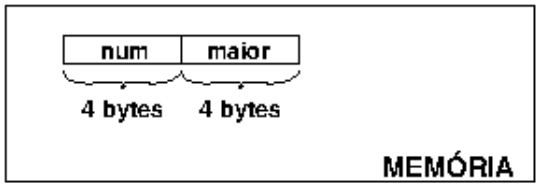

Figura 2.1: Indicativo de como os espaços são alocados na memória.

Ao declararmos uma variável como sendo de um dado tipo fazemos mais do que a simples demarcação sobre o tamanho do espaço e do local que ocupará em memória. Associado a essas operações definimos também quais operações podem ser realizadas sobre aquela variável. De acordo com a complexidade das operações que podem ser realizadas sobre um certo tipo de dado podemos pensar em classificá-los como simples ou estruturados (também chamados compostos). Para os dados simples as operações são usualmente bastante simples e imediatas. O mesmo não ocorre para os dados compostos, para os quais as operações precisam ser especificadas de forma mais complexa.

Antes de examinar as questões relativas às operações sobre dados simples e compostos é importante diferenciar o que é cada um deles, o que é feito a seguir:

- Dados simples são, na realidade, aqueles com quem estamos acostumados a trabalhar, ou seja, inteiros, reais e caracteres;

- Dados compostos são aqueles em que se combinam posições de memória relativas a vários dados simples, como se fossem pertencentes a apenas um nome simbólico (dado), permitindo-se acesso diferenciado entre as várias posições que o compõem.

Ainda para o caso dos dados compostos é possível dividi-los em dois grupos distintos, segundo os tipos de dados simples que estão associados para formar o dado composto. Esses grupos de dados compostos são:

- Estruturas homogêneas, em que os dados simples associados para formar o dado composto são todos do mesmo tipo básico, ou seja, ou são todos reais, ou são todos inteiros, e assim por diante;

- Estruturas heterogêneas, em que os dados associados são de diferentes tipos básicos, formando um novo tipo. As estruturas heterogêneas, por associa-

rem tipos diferentes, têm tratamento mais elaborado do que as homogêneas, o que faz com que seu estudo aconteça separadamente, no capítulo 7.

Nesse capítulo nos ateremos ao exame dos tipos simples e dos compostos homogêneos, apresentando, sempre que oportuno, a sintaxe correspondente em C. Inicialmente apresentamos os tipos numéricos simples, passando para os tipos caracteres simples e compostos, finalizando com os tipos compostos.

2.2 Tipos numéricos

Os tipos numéricos são aqueles com os quais temos trabalhado até o momento, isto é, inteiros e reais. Esses tipos podem ser encarados como básicos dentro de qualquer sistema computacional, e por isso devem ser examinados com um carinho especial.

2.2.1 Inteiros

São variáveis que representam o conjunto dos inteiros ($\mathbb{Z}$). Para ser preciso, temos na realidade apenas um subconjunto de $\mathbb{Z}$, cujo tamanho é determinado pelo tamanho da palavra do computador. Esse tamanho é o número de bits que podem ser armazenados em um único registrador da unidade de processamento (CPU). Por exemplo, temos que processadores mais antigos como o MIPS R3000, usado em estações de trabalho, e versões antigas do pentium, usado em PCs, têm tamanho de palavra igual a 32 bits, o que implica em que os inteiros estão contidos no intervalo $[-2^{31}, +2^{31} - 1]$, enquanto que processadores mais modernos usam 64 bits para representar inteiros, que ficariam então no intervalo $[-2^{63}, +2^{63} - 1]$.

2.2.2 Reais

Como reais podemos representar todo o conjunto $\mathbb{R}$ de números reais, inclusive valores inteiros, desde que sejam representados na forma de um real. A forma de representação de um real exige a presença de um ponto decimal que pode ser deslocado com o uso de potências de dez (por isso operações com reais também são chamadas de operações com ponto flutuante, pois na norma inglesa usa-se o ponto para indicar a parte decimal dos números). A seguir vemos alguns exemplos de reais:

43.877	-1.0	+2.28
-3.14159265	1E-1 (=0.1)	43e+3 (=43000.0) (=43000)

Em particular, a representação de um número real no computador segue um

padrão diferente daquele que estamos acostumados a ver escritos. Como os reais são armazenados na forma de números de ponto flutuante, então os bits que os armazenam são separados em mantissa e expoente. Na prática o número 43.877 seria representado como 43877.10^{-3}. Nele a mantissa é *43877*, enquanto o expoente é *-3*.

2.2.3 Precisão e intervalos de validade em C

Os dois tipos já apresentados tem a precisão e intervalo associados ao tamanho definido para palavra no computador. Entretanto, o C, como também várias outras linguagens, permite que possamos alterar essa precisão se assim desejarmos. Essas alterações de precisão são feitos ou para termos maior precisão numérica ou maior economia de espaço. Assim, num exemplo hipotético (na realidade válido para os antigos processadores com tamanho de palavra de 16 bits, usados até meados da última década do século passado) temos as seguintes precisões:

1. Inteiros:

TIPO	Intervalo de validade	Tamanho em bytes
int	-32768 a +32767	2
unsigned int	0 a 65535	2
short	-128 a +127	1
long int	-2147483648 a 2147483647	4

2. Reais:

TIPO	Intervalo	Tamanho em bytes
float	3.4e-38 a 3.4e+38	4
double	1.7e-308 a 1.7e+308	8

2.2.4 Operadores numéricos

As diversas linguagens de programação permitem várias operações sobre os tipos numéricos, desde que não se misturem tipos conflitantes, isto é, não podemos atribuir um valor real à uma variável inteira, por exemplo. As operações básicas sobre números dentro da linguagem C são nossas velhas conhecidas, sendo compostas basicamente pelas operações aritméticas de **adição, subtração, multiplicação**

e **divisão**, conforme apresentadas a seguir. Além dessas operações, são acrescentadas as operações de **negação unária** (inverte o "sinal" do número, de positivo para negativo e vice-versa), de resto da divisão de inteiros e as de incremento e decremento unitários. Tais operações aparecem resumidas a seguir.

a + b	→	soma dois valores. O resultado é do tipo do operando mais complexo;
a - b	→	subtração. O resultado é do tipo do operando mais complexo;
a * b	→	multiplicação. O resultado segue as regras anteriores;
a / b	→	divisão. O resultado segue as regras anteriores, sendo que se ambos os valores forem inteiros, então o resultado será o quociente da divisão inteira;
a % b	→	resto da divisão inteira de a por b;
-a	→	negação unária, invertendo o sinal de a;
a++	→	incremento unitário de a, sendo aplicável apenas a tipos discretos;
a- -	→	decremento unitário de a, sendo aplicável apenas a tipos discretos;

Além desses operadores a linguagem C permite que se otimize a representação de expressões quando um de seus termos é a mesma variável que receberá a atribuição, como por exemplo na contagem de eventos de um determinado tipo ou acúmulo de notas de uma turma. Nesses casos o que se faz é alterar o operador de atribuição, inserindo nele a operação de incremento ou decremento, como em:

```
TotNotas += nota; // equivale a "TotNotas = TotNotas + nota"
CasosPerd -= Casos; // equivale a "CasosPerd = CasosPerd - Casos"
Fator *= Multiplo; // equivale a "Fator = Fator * Multiplo"
```

Nesses casos, a variável que recebe o valor também faz parte da expressão, aplicando-se o operador que acompanha o sinal de atribuição (=). Como por exemplo, a variável *Fator* recebe na atribuição acima o valor de *Multiplo* multiplicado pelo valor inicial de *Fator*. Assim, se inicialmente temos *Fator=2* e *Multiplo=7*, então a execução do comando resultaria em *Fator=14*.

Na realidade existe uma diferença sutil no uso desses operadores, incluindo-se os de incremento e decremento unitários, no que diz respeito ao momento em que se atualiza os valores das variáveis que recebem as atribuições. Essa diferença pode ser percebida ao usar-se o incremento (ou decremento) unitário dentro de uma expressão. Assim temos:

```
x = b++; // atribui b para x e depois incrementa b
x = ++b; // incrementa b e atribui seu novo valor para x
```

2.2.5 Funções sobre tipos numéricos

Além dos operadores numéricos básicos, a maior parte das linguagens de programação oferece a possibilidade de operações mais complexas sobre os tipos numéricos. Tais operadores envolvem funções tais como as trigonométricas e exponenciais entre outras. Dentro da linguagem C essas funções aparecem, em sua maioria, como parte da biblioteca matemática, com suas definições a partir do arquivo "**math.h**"[2]. Na tabela 2.1 temos algumas dessas funções, com indicações sobre o tipo de operando permitido e do resultado obtido em cada uma delas.

Deve-se salientar que os argumentos usados nessas funções podem, eventualmente, ser valores inteiros. Apenas os resultados devem ser valores reais (em C são do tipo *double*), independentemente do tipo dos argumentos usados. Mais adiante veremos que conversões explícitas de tipo podem contornar isso.

Aqui também deve ficar claro que o uso de funções (matemáticas ou de qualquer outro tipo) simplifica o processo de criação de modelos para o problema a ser resolvido, assim como a geração de seu algoritmo e do código do programa. O próximo exemplo ilustra isso.

EXEMPLO

O cálculo das raízes de uma equação de segundo grau é feito aplicando-se a fórmula de Baskara. Assim, o cálculo dessas raízes usando funções ficaria:

```
....
if ( a != 0 )  // se "a" é diferente de zero executa os comandos
    { delta = b*b - 4*a*c;
      if (delta >= 0) // se "delta" é positivo executa
                          // esses comandos
          { termocomum = sqrt(delta);
            raiz1 = (-b + termocomum) / (2*a);
            raiz2 = (-b - termocomum) / (2*a);
            printf (" Raízes são: %lf e %lf\n", raiz1, raiz2);
          }
       // se "delta" for negativo imprime esta frase
        else  puts (" Não existem raízes reais");
    }
  // se "a" for igual a zero imprime esta frase
  else  printf (" Equação linear com raiz = %lf\n", -c/b);
```

---

[2]Algumas funções, como rand e abs, estão definidas em "stdlib.h".

Tabela 2.1: Algumas funções da biblioteca matemática do C

Função	Descrição	Arg's	Resultado
abs (arg)	valor absoluto (positivo) de arg	int	int
sqrt (arg)	raiz quadrada	real	real
pow (arg1, arg2)	potência ($arg1^{arg2}$)	reais	real
exp (arg)	potência na base e (e^{arg})	real	real
exp10 (arg)	potência na base 10 (10^{arg})	real	real
log (arg)	logaritmo base e de arg (ln^{arg})	real	real
log10 (arg)	logaritmo base 10 de arg (log_{10}^{arg})	real	real
fmod (arg1, arg2)	resto (real) da divisão de $arg1/arg2$	reais	real
sin (arg)	seno (arg em radianos)	real	real
cos (arg)	cosseno (arg em radianos)	real	real
acos (arg)	arco-cosseno ($-1 \leqslant arg \leqslant +1$)	real	real
asin (arg)	arcoseno ($-1 \leqslant arg \leqslant +1$)	real	real
sinh (arg)	seno hiperbólico (arg em radianos)	real	real
cosh (arg)	cosseno hiperbólico (arg em radianos)	real	real
tan (arg)	tangente (arg em radianos)	real	real
ceil (arg)	menor inteiro maior ou igual a arg	real	real
floor (arg)	maior inteiro menor ou igual a arg	real	real
srand (arg)	gera valor aleatório inicial (semente)	real	real
rand ()	gera um número aleatório	—	real
isgreater (arg1, arg2)	retorna valor diferente de zero se $arg1 > arg2$	reais	real
isless (arg1, arg2)	retorna valor diferente de zero se $arg1 < arg2$	reais	real

2.2.6 Precedência entre operadores

As atribuições para *raiz1* e *raiz2* no exemplo anterior nos colocam uma nova questão, que é a ordem na qual cada operação é realizada. Isso é definido, de forma explícita, pelo que chamamos de precedência entre operadores. Na maior parte das vezes (como era de se esperar) a precedência em linguagens de programação segue a ordem de precedência matemática, tal como ocorre em C.

A maior precedência de um operador indica que a operação apontada por ele deve ser realizada antes daquelas apontadas pelos operadores com menor precedência. No caso de operadores de igual precedência será executado antes aquele que aparecer primeiro na expressão, isto é, estiver mais à esquerda (ou próximo do sinal de atribuição).

O uso de parenteses pode alterar a precedência dos operadores, assim como ocorre em expressões aritméticas. Quando temos um par de parênteses, o computador irá executar primeiro a expressão interna ao par, sempre obedecendo localmente as regras usuais de precedência. Apenas depois de concluídas todas as operações internas ao parenteses ele executará o restante da expressão. A tabela 2.2 apresenta a relação de precedência entre os vários operadores.

Tabela 2.2: Ordem de precedência entre operadores em C

OPERADOR	PRECEDÊNCIA
'(', ')', funções intrínsecas	máxima
- (unário), ++, - -	alta
, /, %	média
+, -	baixa

EXERCÍCIO

A partir dos atribuições iniciais abaixo, encontre o valor final da variável x em cada atribuição:

```
{int a, b, c, d, x;
    a = 5;   b = 4;   c = 3;   d = 2;
```

```
1)  x = a*b+c \% d;                    2)  x = a*(b+c) \% d;

3)  x = a*((b+c) \% d);                4)  x = (a*(b+c)) \% d;
```

```
5)  x = a*(b+c \% d);
6)  x = a*(b+c) \% d * c;
7)  x = a*(b+c) \% (d * c);
8)  x = a*((b+c) \% d) * c);
```

2.3 Tipos caracteres

Tipos caracteres são usados para armazenar e manipular informação não-numérica, isto é, dados como nomes de pessoas, endereços, etc. É importante que existam tipos não-numéricos em qualquer linguagem que deseje ter a manipulação simbólica como uma de suas virtudes. Isso é uma demanda importante porque fica difícil imaginar programas que não contenham uma única manipulação simbólica, nem que seja para o uso de chaves em bancos de dados internos ao programa, ou mesmo como simples respostas do tipo "sim" ou "não".

De imediato podem ser imaginados dois tipos diferentes de elementos de tipo caractere. Um em que existam apenas caracteres individuais e outro em que os caracteres estão reunidos em coleções ordenadas, formando palavras ou algo parecido. Os primeiros são tratados simplesmente por caracteres, enquanto que os últimos são conhecidos como ***strings* de caracteres** ou simplesmente ***strings*** ou, em bom português, **cadeias de caracteres**.

Do ponto de vista de memória, variáveis do tipo caractere (individual) ocupam apenas 1 byte, sendo que cada símbolo diferente é representado por códigos especiais, como EBCDIC (em desuso) e ASCII (quase sempre). A codificação em ASCII (*American Standard Code for Information Interchange*) usa os oito bits que formam cada byte para representar até 256 símbolos diferentes. Em ASCII, por exemplo, o dígito '5' é representado pelo byte 00110101, enquanto a letra 'c' é representada por 01100011. Deve-se observar que o dígito '5' é totalmente distinto do inteiro 5, que na memória seria representado pelos bits 00000000000000000000000000000101 (32 bits ao todo).

A forma de tratar tipos caracteres é, na realidade, bastante simples. Basta declarar uma variável como sendo caractere (*char* em C). As atribuições, leituras, comparações, etc., são feitas da mesma forma que os tipos numéricos, sem a necessidade de funções específicas para isso. A ordem relativa entre caracteres (qual aparece primeiro num dicionário de caracteres) obedece a representação binária deles no código ASCII. Assim, temos por exemplo o fato de que o caractere 'Z' ser considerado anterior "alfabeticamente" ao caractere 'a'.

Já as cadeias de caracteres são representadas no computador como sendo uma coleção de caracteres armazenados lado a lado. Esses caracteres podem ser acessados tanto de forma coletiva quanto cada caractere individualmente. Para o acesso de forma individual os procedimentos são iguais aos realizados para caracteres isolados, excetuando-se a necessidade de identificar inequivocamente qual caractere estamos manipulando. Para acesso de forma coletiva é necessário definir funções específicas para cada tipo de procedimento.

2.3.1 Caracteres em C

Em C os caracteres são tratados de forma bastante simples. Um cuidado importante a ser tomado é lembrar que caracteres maiúsculos são diferentes de caracteres minúsculos e, mais ainda, a ordem entre esses caracteres é definida pela sua ordem dentro do código ASCII e não pela ordem alfabética comum.

A declaração de variáveis do tipo caractere segue a seguinte sintaxe:

```
char nomeComDezLetras[11], umDigito, codigoSeisCaracteres[7];
```

Nesse exemplo, a variável *umDigito* é um caractere isolado, ocupando um byte. Já as variáveis *nomeComDezLetras* e *codigoSeisCaracteres* representam coleções com 11 e 7 bytes respectivamente. Deve ser observado que cada cadeia de caracteres deve incluir um símbolo especial para indicar qual é o último caractere da cadeia. Esse símbolo de marcação de fim da cadeia, em C, é o `'\0'`[3]. Assim, as variáveis *nomeComDezLetras* e *codigoSeisCaracteres* podem armazenar no máximo 10 e 6 caracteres respectivamente.

Como já mencionado, a manipulação de caracteres é feita de forma diferente quando temos uma cadeia ou um caractere isolado. No caso de caracteres isolados toda manipulação é feita de forma semelhante ao de tipos numéricos, ou seja, as atribuições são feitas de forma direta. Uma única observação é que quando queremos tratar explicitamente um dado caractere, como atribuir a letra *k* à variável *letra*, devemos colocar esse caractere entre apóstrofos (`'`), como `'0'` ou ainda `letra = 'k'` na atribuição descrita.

Já para as cadeias de caracteres existem funções especiais para seu tratamento. Algumas dessas funções (observando-se que a sintaxe aqui apresentada assume que *str*1 e *str*2 são referências para variáveis do tipo cadeia de caracteres) são:

- `strcmp (str1, str2)` que compara as cadeias *str1* e *str2*, retornando o valor 0 se forem idênticas, um valor negativo se *str1* for lexicograficamente[4]

[3]Ele não é lido a partir do teclado, sendo inserido na cadeia quando o computador entende que a cadeia chegou ao fim, pelo comando *scanf()*

[4]Observar que para os caracteres ASCII a ordem lexicográfica é a ordem em que os caracteres

menor que *str2*, ou um valor positivo caso contrário. É usado em condições de teste como:

```
if (strcmp (palavra, lido) > 0)
       printf("%s vem depois de %s\n",palavra,lido);
  ...
while (strcmp(lido, chave)) ...
```

- `strcpy (str1, str2)` que copia o conteúdo de *str2* em *str1*, como em:

```
strcpy (frase, "Bom Dia");
     ...
strcpy (resposta, frase);
```

- `strcat (str1, str2)` que anexa o conteúdo de *str2* no final de *str1*, como em:

```
strcat(frase, user); // se frase="Bom dia " e user="Garfield"
                     // resultaria em frase="Bom dia Garfield"
```

- `strstr (str1, str2)` que retorna a posição da primeira ocorrência da cadeia *str2* dentro da cadeia *str1*, retornando um valor nulo (0) se ela não for encontrada;

- `strlen (str)` que retorna um inteiro com o tamanho da cadeia *str*, sem contar o símbolo `'\0'`;

Nos exemplos acima aparece o conceito de cadeias de caracteres constantes, como no comando `strcpy (frase, "Bom Dia");`. Da mesma forma que caracteres isolados devem aparecer entre apostrofes, as cadeias de caracteres devem aparecer entre aspas (").

Isso cria o problema de como representar as aspas e o apóstrofo como eles mesmos e não como demarcadores. Em C isso é feito usando-se outro caractere de escape, que é o sinal de barra invertida (\), como em:

```
simb = '\'';       //       ou ainda
strcpy (frase, "Frase com aspas no \" meio");.
```

Outra observação é que como as cadeias de caracteres são, na verdade, coleções de caracteres isolados, podemos aplicar operações de caractere para os componentes da cadeia, desde que identifiquemos cada caractere de forma individual. Isso

aparecem no conjunto ASCII, o que implica, por exemplo, nas letras maiúsculas serem anteriores a qualquer letra minúscula.

é feito usando-se para tanto um índice que diga a posição do caractere na cadeia. A convenção adotada em C para essa identificação marca a primeira posição da cadeia como sendo a posição zero ([0]). Assim, poderíamos escrever comandos como:

```
if (frase[0] = 'A') puts ("Frase começa com A");
...
palavra[4] = '\0';  \\ força o término da cadeia 'palavra'
...
while (str[i] == comp[j]) j++;
```

Como ainda não examinamos comandos de controle (decisão e repetição) não apresentamos aqui exemplos mais elaborados do uso de cadeias de caracteres (nem dos demais tipos básicos). Isso será feito ao longo do capítulo 5, após o exame detalhado de estruturas de decisão e repetição. Assim, poderemos trabalhar com exemplos mais interessantes.

2.4 *Casting* de tipos

Nem sempre os valores com os quais operamos são do mesmo tipo. Isso ocorre em inúmeras situações, como por exemplo quando queremos calcular a raiz quadrada de um valor inteiro, ou ainda quando temos que somar valores rais e atribuir o resultado para uma variável inteira. Enquanto no primeiro caso a solução é trivial, para o segundo isso não ocorre de forma simples. A linguagem C oferece o mecanismo de *casting* para esse tipo de situação. Veremos agora como isso ocorre.

O que é *casting*?

Casting é a operação em que se transforma um valor, armazenado num determinado tipo básico, no seu equivalente em outro tipo básico. Por exemplo, o inteiro 18, armazenado numa palavra de 4 bytes, é armazenado como 00000012H (em hexadecimal para simplificar). Sua conversão para um valor real implica em armazenar-se o valor 18 como mantissa e o valor 0 como expoente.

Como fazer *casting* em C?

O procedimento é bastante simples. Na prática, o compilador já faz isso de forma automática quando a conversão é de um tipo mais simples para um mais complexo

(*int* para *float*, *float* para *double*, etc.). Quando a conversão necessária é de um tipo mais complexo para um mais simples usa-se o que chamamos de *casting* explícito, ou seja, acrescenta-se no comando uma ordem para fazer a conversão de tipos, como em:

```
umaVariavelInteira = (int)(valor_Float * 3.14159265);

diferenca = (int)umchar - (int)outrochar;
```

Outra aplicação importante de *casting* é para se evitar o truncamento na divisão de inteiros. Por exemplo, sabe-se que se as variáveis inteiras $alfa$ e *beta* têm valores 3 e 5 respectivamente, a sua divisão resultará em 0. Isso ocorre mesmo quando o resultado será armazenado numa variável de tipo real (*result*, por exemplo). Para conseguir o resultado correto (0.6 nesse caso) é preciso fazer o *cast* explícito das variáveis inteiras, como em:

```
result = (float)alfa / (float)beta;
```

Outros usos e aplicações de em casting ficarão mais claros quando a complexidade de nossos programas crescer.

EXERCÍCIOS

1. Como valores correspondentes a um dado elemento são representados no programa?
2. O que significa declarar uma variável e o que isso representa na execução de um programa?
3. Como podemos atribuir valores para uma variável do tipo numérico?
4. Como podemos atribuir valores para uma variável do tipo caractere?
5. Porque temos que diferenciar variáveis inteiras de variáveis reais?
6. Qual a diferença entre `"22"` e 22?
7. Revise os comandos para tratamento de cadeia de caracteres.
8. Revise as diferenças de tratamento entre cadeias de caracteres e caracteres isolados.

Capítulo 3

Estruturas de decisão

No primeiro capítulo foram apresentadas duas estruturas de controle, representadas pelos testes de decisão e pelos blocos de repetição. Neste, e no próximo, capítulo iremos examinar com detalhes cada uma das variações possíveis nessas estruturas. Antes, porém, é preciso definir o que é uma estrutura de controle e como esse controle pode ser representado.

Uma boa definição para estruturas de controle é dizer que elas são operações específicas capazes de alterar o fluxo de execução de um programa. A decisão sobre qual caminho o fluxo tomará é feito a partir do exame de uma determinada condição. Numa analogia com um exemplo da vida real, durante a troca de pneu de um carro são tomadas várias decisões. Por exemplo, quando vamos tirar o pneu furado temos que tirar todos os parafusos (ou porcas) que prendem a roda e levantar o carro (não necessariamente nessa ordem!) para então tirar a roda.

A verificação de quais ações podem ser realizadas em um dado momento e como elas ocorrerão é que definem as estruturas de controle em um algoritmo. Por exemplo, na Figura 3.1 pode ser visto o algoritmo para tirar o pneu furado, durante a troca de pneus. O passo 1 é estritamente sequencial e não traz nenhuma forma de controle.

1. Levantar o carro
2. Soltar um parafuso (ou porca)
3. A roda está solta?
   SIM : siga adiante
   NÃO : volte para o passo 2
4. Retirar a roda

Figura 3.1: Algoritmo para tirar uma roda de um carro, ilustrando o uso de estruturas de controle.

No passo 3 desse algoritmo temos uma estrutura de controle que verifica se ainda existem parafusos prendendo a roda. Nesse teste está incluída também a formação de uma estrutura de repetição, fazendo a operação retornar ao passo 2. Deve-se notar que não existe um teste para verificar se o carro foi levantado pois assume-se, implicitamente, que as tarefas são sequenciais, isto é, não podemos iniciar uma tarefa sem que a anterior tenha sido concluída.

Nas próximas seções examinaremos com cuidado todos os aspectos importantes no processo de criação de uma estrutura de controle, tais como a definição das condições de controle examinadas e os tipos de fluxo de execução possíveis.

3.1 Expressões condicionais

As estruturas de controle podem ser entendidas como a composição entre um conjunto de comandos, que representam as ações controladas pela estrutura, e uma expressão condicional, que representa o valor a ser verificado para decidir o caminho a ser seguido. As expressões condicionais são, em geral, formadas por expressões lógicas (verdadeiro ou falso) cujo valor resultante indica o atendimento ou não da condição. Tais expressões são, na verdade, o centro das estruturas de controle e serão examinadas agora.

Como enunciado, uma expressão lógica é uma expressão cujo valor é booleano (verdadeiro ou falso). Assim, estamos interessados em expressões do tipo "A > B", "C = D", etc., para as quais o C (e qualquer outra linguagem) oferece operadores específicos para a sua implementação. Logo, para entender a sintaxe das expressões lógicas devemos aprender quais são esses operadores, o que é nosso próximo passo.

3.1.1 Operadores lógicos

1. Operadores *booleanos*

   São operadores sobre expressões lógicas, também chamadas *booleanas*, que permitem alterar valores *booleanos* e associa-los em expressões mais complexas. Temos três tipos básicos de operadores *booleanos*:

   (a) Operador de negação (NOT), que faz a negação do valor *booleano* de uma expressão, isto é, inverte o seu valor original.

   (b) Operador de conjunção, em que o resultado da sua aplicação sobre duas expressões é um valor *booleano* segundo a tabela-verdade apresentada na tabela 3.1:

Tabela 3.1: Tabela-verdade do operador de conjunção

expressão 1	*expressão 2*	*RESULTADO* (AND)
falso	falso	falso
falso	verdadeiro	falso
verdadeiro	falso	falso
verdadeiro	verdadeiro	verdadeiro

Por essa tabela vemos que o resultado de uma operação de conjunção só será verdadeiro quando todas as expressões envolvidas forem verdadeiras. Basta que uma delas seja falsa para que a expressão RESULTADO também seja falsa.

(c) Operador de disjunção, na qual o resultado será verdadeiro quando pelo menos uma das expressões sobre as quais é aplicado for verdadeira, como mostra a tabela-verdade da tabela 3.2.

Tabela 3.2: Tabela-verdade do operador de disjunção

expressão 1	*expressão 2*	*RESULTADO* (OR)
falso	falso	falso
falso	verdadeiro	verdadeiro
verdadeiro	falso	verdadeiro
verdadeiro	verdadeiro	verdadeiro

2. Operadores relacionais

São operadores sobre expressões não booleanas cujo resultado é um valor booleano. Tais operações verificam relações entre valores diversos e retornam valores booleanos indicando se uma dada relação é ou não verdadeira. Na lista a seguir apresentamos os operadores relacionais matemáticos. Vale observar que cada linguagem apresenta os seus próprios símbolos para representar esses operadores. No caso de C apresentamos ao final desta seção a relação completa de todos os operadores relacionais e lógicos, assim como a sintaxe adotada na implementação das estruturas de controle.

(a) *expr1* $>$ *expr2* , que retorna "verdadeiro" se o valor de *expr1* for maior que o valor de *expr2*. Caso isso não ocorra o valor retornado é "falso".

(b) *expr1* $\geqslant$ *expr2* , que retorna "verdadeiro" se o valor de *expr1* for maior que ou igual ao valor de *expr2*. Caso isso não ocorra o valor retornado é "falso".

(c) *expr1* $<$ *expr2* , que retorna "verdadeiro" se o valor de *expr1* for menor que o valor de *expr2*. Caso isso não ocorra o valor retornado é "falso".

(d) *expr1* $\leqslant$ *expr2* , que retorna "verdadeiro" se o valor de *expr1* for menor que ou igual ao valor de *expr2*. Caso isso não ocorra o valor retornado é "falso".

(e) *expr1* $=$ *expr2* , que retorna "verdadeiro" se o valor de *expr1* for igual ao valor de *expr2*. Caso isso não ocorra o valor retornado é "falso".

(f) *expr1* $\neq$ *expr2* , que retorna "verdadeiro" se o valor de *expr1* for diferente do valor de *expr2*. Caso isso não ocorra o valor retornado é "falso".

3.1.2 Combinando expressões lógicas

Todos os operadores apresentados até agora podem ser combinados de forma a obtermos expressões mais complexas, que retornam num único valor a combinação de vários resultados intermediários. Assim podemos, por exemplo, descobrir se um número é maior que zero e menor que 100 numa única expressão, permitindo a redução do número de expressões no programa e, portanto, melhorando sua legibilidade. A seguir temos alguns exemplos do uso dessas combinações:

(x $>$ z) AND (y $>$ z)

(x + y/z) $\leqslant$ 3.5

NOT flag OR (y $>$ z)

NOT (flag OR (y $>$ z))

EXERCÍCIO:

Verifique o valor de cada uma das expressões dos exemplos anteriores, assumindo os valores a seguir para suas variáveis:

a) x = 3, y = 4, z = 3, flag = TRUE

b) x = 2, y = 2, z = 1, flag = FALSE

c) x = 1, y = 4, z = 5, flag = TRUE

3.1.3 Operadores lógicos e relacionais em C

Cada linguagem de programação usa diferentes símbolos para representar os operadores relacionais e lógicos. A estrutura funcional desses operadores é a mesma

em todas as linguagens, bastando que se tenha a compreensão do que significa cada operador. Em específico para a linguagem C temos os seguintes operadores:

A Operadores lógicos -

a. Negação	$\longrightarrow$	**!**	, como em	*! zero*
b. Conjunção	$\longrightarrow$	**&&**	, como em	*valor1 && valor2*
c. Disjunção	$\longrightarrow$	||	, como em	*valor1 || valor2*

B Operadores relacionais -

a. Maior que	$\longrightarrow$	$>$	, como em	*valor1 > valor2*
b. Menor que	$\longrightarrow$	$<$	, como em	*valor1 < valor2*
c. Maior ou igual	$\longrightarrow$	$>=$	, como em	*valor1 >= valor2*
d. Menor ou igual	$\longrightarrow$	$<=$	, como em	*valor1 <= valor2*
e. Igual	$\longrightarrow$	$==$	, como em	*valor1 == valor2*
f. Diferente	$\longrightarrow$	**!=**	, como em	*valor1 != valor2*

Ao longo das próximas seções e capítulos o uso desses operadores deve ficar mais claro. Entretanto, é preciso fazer aqui uma breve descrição de como o C trata os resultados desses operadores.

Na realidade para o C não existem os valores lógicos da forma VERDADEIRO ou FALSO. Assim, em C um valor é assumido como sendo falso se for igual a zero (ou nulo para tipos não-numéricos). Qualquer valor diferente de zero (ou nulo) é considerado como sendo verdadeiro.

Uma implicação imediata dessa condição é que qualquer variável pode ser testada diretamente, sendo o resultado do teste positivo se a variável tiver valor diferente de zero (qualquer que seja ele).

3.2 Estruturas de decisão simples

A estrutura de controle mais usada é o teste de decisão. Isso porque um teste decisão é aquele que verifica se uma determinada condição é satisfeita e decide, a partir desse resultado, qual de dois caminhos (ou ramos) será seguido. A ideia básica aqui funciona na forma "Se tal coisa for verdade então faça isso, senão faça algo diferente disso".

Existem, na maioria das linguagens estruturadas, dois tipos de estruturas de decisão. Uma que opera a partir de um único valor, decidindo portanto entre dois caminhos possíveis (atende ou não atende), e outra que opera sobre valores discretizados (enumerados) de uma variável. Examinaremos primeiro o caso mais simples, de dois caminhos, ampliando então sua complexidade e escopo de aplicação até chegar-se ao segundo caso.

Em geral, um teste de decisão simples pode ser construído de duas formas. A diferença entre essas formas está em como o caminho alternativo (quando a condição testada é falsa) é construído. Assim temos:

- Teste sem caminho alternativo –

  Aqui o teste faz a escolha entre executar um conjunto de comandos, quando a condição é verdadeira, e não fazer nada quando é falsa. Num algoritmo essa construção ficaria:

  SE *condição a ser testada*
  ENTÃO *comandos executados se a condição for verdadeira*

- Teste com caminho alternativo –

  Nesse caso executa-se um certo conjunto de comandos quando a condição testada é verdadeira. Caso contrário executa-se um outro conjunto de comandos. Num algoritmo teríamos:

  SE *condição a ser testada*
  ENTÃO *comandos executados se a condição for verdadeira*
  SENÃO *comandos executados se a condição for falsa*

3.2.1 Testes de decisão simples em C

O teste de decisão simples é implementado na linguagem C através do comando **if**, cuja sintaxe pode ser vista na seguinte estrutura:

if (*condição testada* **)**
comandos executados se a condição for verdadeira

ou ainda

if (*condição testada* **)**
    comandos executados se a condição for verdadeira
else
    comandos executados se a condição for falsa

Os pequenos programas das figuras 3.2 e 3.3 ilustram mais claramente como esse comando pode ser utilizado em C. O primeiro deles determina qual é o maior entre dois números lidos. Nesse programa é um exemplo de uso do *if* com apenas a parte do *então*.

```
#include "stdio.h"

main ( )
{int primeiro, segundo, maior_deles;

   puts("Digite o primeiro número");
   scanf ("%d", &primeiro);
   maior_deles = primeiro; /* assume inicialmente que primeiro
                              é o maior dos dois números     */
   puts("Digite o segundo número};
   scanf("%d", &segundo);
   if (segundo > primeiro)    // testa se segundo é maior que
                              // primeiro e troca maior_deles se
       maior_deles = segundo; // for verdade
   printf (" O maior dos números é %d\n", maior_deles);
}
```

Figura 3.2: Exemplo do uso do comando *if*

Já o segundo programa verifica se dois números possuem sinais diferentes e usa o *if* com *então-senão*. Se os número tiverem sinais diferentes eles serão multiplicados. Se tiverem mesmo sinal eles serão somados.

No caso de termos que executar mais do que um comando dentro de um dos ramos de um *if* temos que colocar todos os comandos dentro de um par de chaves (**{ ... }**). Por exemplo, o programa da figura 3.3 poderia ser alterado para que a mensagem da resposta fosse mais significativa, como visto na Figura 3.4.

```
#include "stdio.h"

main ( )
{int res, num1, num2;

  puts("Digite dois números inteiros");
  scanf("%d %d", &num1, &num2);
  if (num1 * num2 < 0) // verdadeiro se sinais forem diferentes
      res = num1 * num2;
    else
      res = num1 + num2;

  printf (" O resultado final é %d \n", res);
}
```

Figura 3.3: Exemplo do uso do comando *if-then-else*.

```
#include "stdio.h"

main ( )
{int res, num1, num2;

  puts("Digite dois números inteiros");
  scanf("%d %d", &num1, &num2);
  if (num1 * num2 < 0) // verdadeiro se sinais forem diferentes
      { res = num1 * num2;
        printf (" O produto desses números é %d \n", res);
      }
    else
      { res = num1 + num2;
        printf (" A soma desses números é %d \n", res);
      }
}
```

Figura 3.4: Exemplo de *if-then-else* com vários comandos nos ramos.

3.2.2 Encadeamento de testes de decisão

Testes de decisão não precisam aparecer de forma isolada. Dentro dos comandos executados num teste de decisão podemos ter um outro teste de decisão. Quando

isso ocorre dizemos que o segundo teste está aninhado no primeiro. Esses aninhamentos podem ser arranjados de acordo com as necessidades do programa, sem nenhuma restrição adicional.

O único cuidado a ser tomado quando temos testes aninhados é o emparelhamento dos respectivos ramos *então* e *senão*. Por exemplo, no trecho a seguir qual seria o valor da variável A no seu final?

```
A = 5;   B = 0;   C = -5;
if (B > 0)
if (C > 0)
A = B;
else
A = C;
printf("%d\n",A);
```

A pergunta a ser respondida para se obter o valor correto é "a qual *if* pertence o ramo *else*?" Da forma como esse trecho está escrito não podemos afirmar (ou demonstrar) a qual ramo o *else* pertence. Isso ocorre por uma falha na especificação da gramática do C (e de várias outras linguagens) e é um problema conhecidíssimo em teoria da compilação[1].

Felizmente, a solução (prática) para o problema de ambiguidade do teste de decisão também é conhecida e diz que um *else* sempre se refere ao *if* mais próximo que ainda não tenha um *else* associado, exceto quando existirem chaves delimitando os comandos. Assim, o valor final de A no nosso exemplo é 5.

Assim, se no mesmo exemplo quiséssemos que o *else* se referisse ao primeiro *if*, caso o modelo de resolução do problema assim exigisse, teríamos que acrescentar um par de chaves cercando o comando *if* mais interno e seu único comando ($A = B$;), como em:

```
A = 5;   B = 0;   C = -5;
if (B > 0)
    { if (C > 0)
        A = B;
    }
  else
    A = C;
printf("%d\n",A);
```

[1]Na prática a "falha" é intencional, uma vez que ao deixá-la os projetistas dessas linguagens procuraram diminuir o número de símbolos e regras da linguagem e existe uma forma simples de contornar a ambiguidade.

Nesse código, além de usarmos a indentação para deixar claro o que queremos, aparece explicitamente o par '{' e '}', indicando que o segundo comando *if* começa e termina entre o par de chaves. Isso, automaticamente, faz com que o *else* esteja ligado ao primeiro *if*, resultando em **A = -5**.

3.3 Estrutura de decisão de múltipla escolha

Quando queremos fazer uma sequência de testes relacionados e que atuem apenas em uma variável podemos seguir dois caminhos distintos. Num deles fazemos uma cadeia de estruturas de decisão (aninhadas para evitar testes cujos resultados já seriam previamente conhecidos), na forma *if-then-else-if-then-...-else*, ou podemos usar os chamados testes de múltiplas escolhas, que são disponíveis em várias linguagens de programação.

A primeira alternativa (encadeamento de testes de decisão) é mais imediata e aparentemente simples. Porém, nem sempre essa é a solução mais elegante do ponto de vista de legibilidade do programa. Suponha por exemplo que queremos fazer um programa que leia um dígito entre 0 e 9 e execute uma ação diferente para cada valor distinto. Se usássemos testes de decisão aninhados teríamos algo do tipo visto no algoritmo da figura 3.5, que é um tanto quanto difícil de ser compreendido.

```
Se num = 0
  então faça ACAO_ZERO
  senão Se num = 1
          então faça ACAO_UM
          senão Se num = 2
                   então faça ACAO_DOIS
                   senão Se num = 3
                            então faça ACAO_TRES
                          .....
```

Figura 3.5: Múltiplas escolhas com encadeamento de testes de decisão

Com teste de decisão de múltipla escolha aglutinamos todas as diferentes opções em um único comando, como é visto na figura 3.6. O funcionamento de um comando desse tipo é, na realidade bastante simples. Examina-se inicialmente o valor da variável de controle (*num*, nessa figura), a qual deverá possuir valores discretos (ou enumeráveis) para que se possa enumerar uma série de ca-

sos distintos. Em havendo um dos casos em que o valor da variável de controle é equivalente ao valor definido, então a ação deste caso será executada e o comando encerrado. Se nenhum caso combinar com o valor da variável, então nada será executado ou, em certas condições, um caso padrão (*default*) será executado.

```
Caso num
   for igual a 0 então ACAO_ZERO
   for igual a 1 então ACAO_UM
   for igual a 2 então ACAO_DOIS
   for igual a 3 então ACAO_TRES
   .....
Fim de Caso
```

Figura 3.6: Múltiplas escolhas com testes de decisão de múltipla escolha

3.3.1 Testes de decisão múltipla em C

No caso da linguagem C, a sintaxe para testes de decisão de múltipla escolha é a seguinte:

```
switch  ( variável_discreta )
{ case valor1 : ação_valor1; break;
  case valor2 : ação_valor2; break;
  ...
  default:  ação_default;
}
```

A linha com o caso *default* é opcional, sendo usada para definir uma ação que será executada caso nenhuma outra opção satisfizer o valor de *variavel_discreta*.

Aparece também nesse exemplo o comando *break*, que é usado para delimitar quais ações serão executadas para um determinado valor de *case*. Assim, em C é possível definir que determinados comandos serão executados para mais do que um *case*, como em:

```
y = 0;
switch  ( x )
{ case 1 : y = 1;
  case 2 : y = y + 1; x = y; break;
  case 3 : x = y; break;
}
```

Nesse exemplo, caso o valor da variável x seja 3, executa-se apenas o comando *"x = y;"*, resultando em *x=0*. Da mesma forma, se x for igual a 2, então executam-se os comandos do *case 2*, resultando em *x=1*. Finalmente, se o valor inicial fosse 1, então executariam-se os comandos dos *case 1* e *case 2*, em sequência, pois não existe um *break* entre eles. Isso resultaria em *x=2*. Deve-se observar ainda que podemos ter ramos *case* vazios, sem nenhum comando executado, com ou sem o *break*, dependendo do que deve ser resolvido.

3.3.2 Aninhamento de testes de decisão múltipla

Da mesma forma em que aninhamos testes de decisão simples, podemos fazer o aninhamento de testes de múltipla escolha. O aninhamento deste tipo de teste é útil em situações em que a ação a ser executada não depende apenas do valor de uma única variável. Assim, poderemos construir soluções em que primeiro selecionamos algo a partir de uma chave primária (estado de um sistema, por exemplo) e depois fazemos a seleção a partir de uma chave secundária (ação desejada sobre o sistema). Isso permite trechos de programa como o algoritmo da figura 3.7.

```
Caso EstadoDoSistema
  for igual a Em_espera então
     Caso AcaoDesejada
       for igual a Desligue então ACAO_DESLIGAR
       for igual a Trabalhe então ACAO_TRABALHAR
     Fim de caso
  for igual a Desligado então
     Caso AcaoDesejada
       for igual a Trabalhe então ACAO_LIGAR
       for igual a Desligue então ACAO_NADA
     Fim de caso
  for igual a Trabalhando então
     Caso AcaoDesejada
       for igual a Trabalhe então ACAO_NADA
       for igual a Desligue então ACAO_DESLIGAR
     Fim de caso
Fim de caso
```

Figura 3.7: Uso de testes de decisão de múltipla escolha aninhados.

Uma outra situação em que o aninhamento desse tipo de teste é interessante ocorre em servidores de múltiplas finalidades. Nesse caso, o teste mais externo ve-

rificaria qual a finalidade do serviço requerido, enquanto o segundo teste decidiria como o serviço seria executado.

Já que estamos falando em exemplos, uma última porém extremamente importante aplicação é o de reconhecimento de objetos em uma frase. Essa é a primeira operação executada em um compilador, identificando nomes de variáveis, constantes, palavras reservadas, etc. Normalmente isso pode ser implementando com um teste múltiplo sobre qual o carácter que acabou de ser lido e, para cada caso possível faz-se um novo teste múltiplo sobre qual o estado anterior da leitura.

Em qualquer situação o cuidado a ser tomado é não misturar os testes. Isso implica em que os casos do teste externo devem estar num mesmo nível e cada teste interno deve ter seus casos nesse nível mais interno.

3.4 O comando de decisão ternária

Embora os comandos de decisão já descritos sejam suficientes para tratar qualquer problema computacional, algumas linguagens incluem um comando de decisão ternária, em que em apenas um comando se faz o teste e as atribuições para os casos do teste resultar em verdade ou falso. No caso de C a sintaxe do comando de decisão ternária é dado por:

```
( teste ) ? comando se verdade : comando se falso ;
```

Os caracteres '?' e ':' server como separadores entre as três partes do comando. Os comandos a seguir exemplificam sua aplicação:

```
( b > c ) ? a = 0 : (a = 1);
printf("i=%d, que é %s\n",i,(i%2) ? "ímpar" : "par");
```

O primeiro comando atribui 0 para a variável `a` caso `b` seja maior que `c`, ou 1 caso não seja. Já o segundo comando imprime "ímpar" se o valor de `i` for ímpar e "par" caso contrário.

Deve ficar claro que os efeitos desse comando podem ser obtidos com comandos `if-then-else`. No caso dos comandos apresentados no exemplo a substituição fica assim:

```
// (b>c)? a=0 : (a=1);
if ( b > c )
    a = 0;
  else
    a = 1;
```

```
// printf("...",i,(i%2)?...);
if (i%2)
    printf(" ... ímpar\n",i);
  else
    printf(" ... par\n",i);
```

Capítulo 4

Estruturas de repetição

As estruturas de controle de decisão vistas no capítulo anterior são bastante úteis para definir fluxos de instruções que serão executadas apenas uma vez. Isso, entretanto, é insuficiente na maioria das vezes. No nosso algoritmo da troca de pneu 3.1, por exemplo, é necessário repetir a pergunta se todas as porcas já foram retiradas por várias vezes. Isso nos leva às estruturas de controle de repetição, as quais serão examinadas neste capítulo, partindo do princípio de que em algumas situações já sabemos quantas vezes uma determinada operação será repetida (como o caso da troca de pneu) e em outras isso não pode ser previamente determinado (como jogar um dado até sair o número seis).

4.1 Estrutura de repetição enumerável

Existem situações em que se deseja executar um grupo de comandos de forma repetitiva. Uma solução seria repetir, no código do programa, as instruções correspondentes quantas vezes quiséssemos executá-las. Entretanto isso não é nada inteligente pois cada vez que quiséssemos variar esse número de repetições teríamos que reescrever e recompilar o programa. Para evitar isso existem estruturas de controle que nos permitem repetir um grupo de instruções tantas vezes quanto desejarmos, através de um simples teste para verificar se as instruções devem ou não ser executadas mais uma vez. Esse tipo de estrutura forma um bloco denominado **laço de repetição** (ou simplesmente *loop* em inglês).

Esses laços podem ser construídos de diferentes maneiras, dependendo da forma como o teste de verificação é aplicado. Esse teste de verificação é conhecido como condição de parada do laço, e funciona da mesma forma que os testes de decisão simples examinados na seção 3.2. O que diferencia os vários tipos de blocos de repetição é o momento em que o teste é aplicado e a maneira como o valor da variável de controle é atualizado. Em particular, quando o número de

repetições pode ser conhecido *a priori*, quer seja através de uma constante ou pelo valor de uma variável que permaneça inalterado por algum tempo, podemos usar estruturas de repetição enumeráveis, como a definida a seguir:

Para valores de Controle **indo de** *valor_inicial* **até** *valor_final*,
com incremento/decremento de *tanto*
Faça *Comandos do corpo da repetição*

Em que **Controle** é a variável que enumera as repetições dos comandos da estrutura, sendo limitada pelas constantes *valor_inicial* e *valor_final*. A forma como o valor da variável de controle de repetição evolui de um limite ao outro varia de acordo com a necessidade da aplicação, podendo ser através de incrementos ou decrementos de valores pré-determinados, definidos pelo valor de *tanto* no comando acima.

Como dito no início desta seção, o uso de estruturas enumeráveis é possível sempre que temos condições de saber quantas repetições executaremos no momento em que entramos no laço. Embora isso nem sempre ocorra, existe uma infinidade de situações em que podemos usar essas estruturas. A vantagem no uso delas é que o código fica mais claro ao especificar quantas vezes o corpo do laço será executado.

Situações em que isso ocorre incluem aplicações sobre matrizes (aquelas vistas em matemática), conjuntos de elementos de tamanho conhecido e várias outras situações similares. A dica sobre quando usar uma estrutura de repetição enumerável é verificar no modelo do programa (ou em seu algoritmo em mais alto nível) se os comandos a serem repetidos podem ser contabilizados antes de serem executados e se a variável que controlará a repetição pode ser discretizada para valores pré-demarcados. Sempre que isso ocorrer usaremos preferencialmente essas estruturas.

No capítulo 1 resolvemos o cálculo do fatorial através de um outro tipo de estrutura de repetição. Na prática as estruturas enumeráveis se prestam bastante bem para casos como o do cálculo do fatorial. Assim, aquele programa pode ser modificado para conter um laço enumerável na sua implementação, como visto na figura 4.1.

EXERCÍCIO

Escreva um algoritmo para um programa que calcule a tabela de conversão de graus celsius para fahrenheit num intervalo determinado por dois limites dados por um usuário. Imprima o valor em celsius e seu equivalente em fahrenheit.

```
.... scanf("%d",&num);
     if (num < 0)
         { puts (" Não existe fatorial de números negativos");
           exit (0);
         }
     if ((num == 0) || (num == 1))
         printf (" Fatorial de %d = 1\n", num);
       else
         { fatorial = 1;
           for (i=num; i > 1; i- -)
               fatorial = fatorial * i
           printf (" Fatorial de %d = %d\n", num, fatorial);
         }    ....
```

Figura 4.1: Fatorial usando estruturas de repetição enumerável

4.2 Estruturas de repetição não-enumeráveis

Estruturas enumeráveis funcionam bem quando o número de repetições é conhecido antes do início do laço. Entretanto, nem sempre isso é possível, ou seja, muitas vezes queremos repetir um conjunto de comandos até que uma dada condição seja satisfeita, sendo que esta condição depende de ações internas ao laço.

Por exemplo, se num dado programa desejarmos ler números e calcular o fatorial de cada um deles até cansarmos dessa brincadeira, temos que encontrar uma maneira de fazer com que o computador saiba quando deve parar. Isso pode ser feito através de uma variável de controle, como o número digitado por nós. Assim, quando digitarmos um número negativo, por exemplo, estaremos dizendo ao computador "PARE". Para fazer isso precisamos de uma estrutura de repetição não-enumerável. Esta estrutura pode ser representada em algoritmos pelo comando **Enquanto**, já visto no exemplo do fatorial no capítulo 1.

Estruturas não enumeráveis podem assumir duas formas distintas de execução, dependendo do momento em que se examina a condição de controle do laço. Numa das formas o corpo do laço é executado sempre antes de se verificar se a condição já foi atendida. Na outra forma a condição é verificada antes e, caso seja atendida, executa-se o corpo do laço. Tais formas seriam representadas, num

algoritmo, com a seguinte sintaxe:

Teste antecipado

Enquanto *condição* **for verdade**
Faça *comandos do corpo do laço*

Teste posterior

Faça *comandos do corpo do laço*
Enquanto *condição* [**não**] **for verdade**

Em especial, na estrutura para o teste posterior, é possível duas construções distintas. Uma em que se repete o conjunto de comandos do corpo do laço até que a condição se torne verdadeira, como é o caso do comando "repeat-until" da linguagem Pascal. Já em outra construção o corpo é repetido enquanto a condição for verdadeira, como é o caso do comando "do-while" do C.

EXEMPLO

Suponha que temos que escrever um programa que examine um conjunto de números, até que encontre um número primo. Um algoritmo (em alto nível) para isso é visto a seguir:

```
1. LEIA um número
2. ENQUANTO número não for primo FAÇA
   2.1 LEIA um número
3. Escreva "Número primo foi encontrado"
```

Nesse exemplo fica fácil de observar que seria mais simples usar a estrutura que permite o teste posterior, uma vez que antes de fazer o teste temos que ler o número de qualquer forma e essa é a única operação realizada dentro do corpo do laço de repetição. A troca da estrutura resultaria no seguinte algoritmo:

```
1. FAÇA
   1.1   LEIA um número
   ENQUANTO número não for primo
2. Escreva "Número primo foi encontrado"
```

4.3 Estruturas de repetição em C

As estruturas definidas nas páginas anteriores estão representadas numa linguagem mais algorítmica, sem uma maior dependência para uma ou outra linguagem. Todas as linguagens de programação apresentam algum mecanismo para a implementação dessas estruturas de repetição. No caso da linguagem C temos três diferentes comandos para realizar a implementação de laços de repetição estruturados (existem laços não-estruturados, implementados de uma outra forma, mas que devem ser evitados em um programa bem definido).

A sintaxe dos comandos em C que implementam laços de repetição são vistos a seguir. Deve ficar claro que exemplos da aplicação desses comandos serão vistos na próxima seção (bem como em vários outros problemas ao longo do restante deste texto).

4.3.1 Repetição enumerável - o comando *for*

O comando **for** permite a construção de laços de repetição enumeráveis, isto é, com um controle automático no total de repetições a serem executadas. A sintaxe desse comando é vista a seguir:

for (*origem* ; *condição_limite* ; *forma_de_avanço*)
{ *Corpo do laço* **}**

Em que:

origem	é a definição dos valores iniciais para as variáveis que servirão como controle do laço;
condição_limite	define o critério de parada para as repetições do corpo do laço;
forma_de_avanço	define a forma de atualização das variáveis de controle a cada iteração do laço;
Corpo do laço	representa o comando ou conjunto de comandos que serão executados a cada iteração do laço, sendo que quando forem dois comandos ou mais é preciso agrupa-los através de um par de chaves ({ ... }).

4.3.2 Repetição não-enumerável - O comando *while*

O comando **while** permite a construção de laços de repetição em que o teste para continuar ou não a execução do laço é feito antes de se executar o corpo do laço. A sintaxe desse comando é vista a seguir:

while (*exp_controle*)
{ *Corpo do laço* **}**

Em que:

exp_controle	define o critério de parada para as repetições do corpo do laço, através de uma expressão lógica;
Corpo do laço	representa o comando ou conjunto de comandos que serão executados a cada iteração do laço, sendo que quando forem dois comandos ou mais é preciso agrupa-los através de um par de chaves ({ ... }).

4.3.3 Repetição não-enumerável - O comando *do-while*

O comando **do-while** é usado para a construção de laços em que se tenha que executar o corpo antes de se verificar se essa execução deve continuar ou não. Sua sintaxe é apresenta a seguir.

do
{ *Corpo do laço* **}**
while (*exp_controle*)

Em que:

exp_controle	define o critério de parada para as repetições do corpo do laço, através de uma expressão lógica;
Corpo do laço	representa o comando ou conjunto de comandos que serão executados a cada iteração do laço, sendo que quando forem dois comandos ou mais é preciso agrupa-los através de um par de chaves ({ ... }).

OBSERVAÇÃO

Como já dito na seção anterior, os comandos de repetição não-enumeráveis diferem apenas na forma em que a expressão de controle (*exp_controle*) é considerada. No comando **while** a execução do laço ocorre apenas se o valor dessa expressão for verdadeiro (ou seja, diferente de zero ou nulo). Já no comando **do-while** o corpo do laço é executado pelo menos uma vez, sendo repetido enquanto o valor da expressão for verdadeiro.

4.4 Exemplos no uso de laços de repetição

A figura 4.2 apresenta exemplos bastante simples do uso desses tipos de estruturas de repetição disponíveis em C. Nos três laços o objetivo é fazer a soma dos primeiros *num* inteiros. Como esse é um problema enumerável, sua solução é mais simples usando o laço **for**, em que basta usar a variável de controle do laço (i) como elemento da soma. Nos laços **while** e **do-while** é preciso incrementar explicitamente essa variável a cada iteração executada.

```
#include "stdio.h"

main()
{int num, i, soma;

  scanf("%d",&num);
  soma = 0;
  for (i=0; i <= num; i++)
       soma = soma + i;  // Como se trata de apenas um comando
                         // não é preciso o uso de chaves
  printf (" A soma pelo for resulta em %d\n", soma);
  soma = 0;
  i = 1;
  while (i <= num)
  { soma = soma + i;
    i++;  // AVANÇO DO CONTROLE
  }
  printf (" A soma pelo while resulta em %d\n", soma);
  soma = 0;
  i = 1;
  do
  { soma = soma + i;
    i++;  // AVANÇO DO CONTROLE
  } while (i <= num);
  printf (" A soma pelo do-while resulta em %d\n", soma);
}
```

Figura 4.2: Exemplo genérico de uso de laços de repetição

A diferença principal entre os comandos **for** e os **while** e **do-while** está na forma de avanço do valor da variável de controle. No caso do **for** esse avanço é

indicado diretamente no comando, como seu terceiro parâmetro. Já os comandos **while** e **do-while** não apresentam esse avanço explícito. Assim, o programador deve introduzir no corpo do laço alguma forma de avanço sobre a variável de controle. No programa da figura 4.2 isso é feito pelos comandos *i++* marcados pelos comentários "`AVANÇO DO CONTROLE`".

O uso de cada uma das estruturas de repetição deve ser criterioso, isto é, devemos sempre tomar o cuidado de escolher a estrutura mais adequada para nossos objetivos para o corpo do laço. O mesmo pode ser dito com relação às estruturas de decisão já estudadas (**if** e **switch**). No caso do **do-while** deve ficar claro que o usaremos sempre que for necessária a execução de uma determinada ação pelo menos uma vez antes que se saia do laço.

Um caso em que isso ocorre é a leitura de um arquivo de dados sequencial, em que os dados são lidos um após o outro até o final do arquivo. Existe em C uma função que verifica se foi lido o final de arquivo, que é a **feof (leitor_arq)**. Entretanto, é necessário primeiro que se tente ler algo no arquivo para saber se o mesmo está ou não no seu final. Assim, um trecho de programa para leitura de arquivo ficaria assim:

```
do
{ fscanf (arq, "......", .......);
// fscanf equivale ao scanf, só que para ler arquivos
    ...
} while ( ! feof (arq) )
```

Examinemos agora outras situações em que seja necessário repetir ações.

Caso 1 - considere que o seu problema é fazer a soma dos elementos de dois vetores de tamanho *n*, guardando o resultado em um terceiro vetor. Como a operação a ser realizada deve ser repetida um número fixo de vezes, então é evidente que a melhor solução é usar o comando **for**. O trecho de código a seguir ilustra essa aplicação.

```
for (i=0; i<n; i++) // faz n somas...
     vet3[i] = vet1[i] + vet2[i];
```

Caso 2 - considere que o seu problema é encontrar um número ímpar em um vetor de tamanho *n*, parando quando ele for encontrado. Como a operação a ser realizada pode ser repetida uma ou mais vezes, com o máximo de *n* vezes, então é evidente que a melhor solução é usar os comandos **while** ou **do-while**. O trecho

de código a seguir ilustra essa aplicação com o uso do **while**.

```
i = 0;
while ((i < n) && (vet[i]%2 == 0)) // o teste i<n serve como
       i++;                        // limite superior do laço
if (i < n)  printf("%d é ímpar\n",vet[i]);
// esse último teste é verdade apenas se o comando while
// terminou porque foi encontrado um número ímpar !!
```

Caso 3 - considere que o seu problema é encontrar o maior valor armazenado em um vetor e apresentá-lo para o usuário. Mais uma vez aqui temos que repetir a operação de examinar o conteúdo de uma posição do vetor um número fixo de vezes. Para cada uma delas temos que ver se o atual maior número encontrado ainda é maior que o valor examinado. Se sim, continuamos a busca, se não fazemos uma atualização no valor do maior número. Então, como o número de repetições é fixo devemos usar o comando **for**. O trecho de código a seguir ilustra essa aplicação.

```
maior = -NUMGRANDE; // algum valor negativo muito grande
for (i=0; i<n; i++) // faz n somas...
{ if (vet[i] > maior)
      maior = vet[i];
}
printf ("maior = %d\n",maior);
```

4.5 Cuidados com laços de repetição

Laços de repetição formam um mecanismo bastante poderoso em computação. Entretanto, alguns cuidados devem ser tomados para que os mesmos funcionem da forma esperada. Esses cuidados envolvem basicamente a condição de parada do laço, ou seja, não parar antes ou depois do ponto em que se deve parar. Uma lista desses cuidados é apresentada a seguir:

- Ao trabalhar com vetores e matrizes deve-se tomar o cuidado de que o laço não ultrapasse seus limites. Lembrar aqui que o primeiro elemento tem índice *0* e se a estrutura possui *k* elementos, então se deve parar na posição de índice *k-1*.

- Em qualquer situação é preciso lembrar de colocar uma condição de parada limite, evitando laços infinitos (a menos que a aplicação exija essa condição).

EXERCÍCIO

Refazer o programa fatorial usando **do-while** no lugar de **while**.

4.6 Aninhamento de estruturas de repetição

Até agora vimos apenas estruturas de repetição isoladas. Entretanto, muitas vezes precisamos fazer laços contendo outros laços, isto é, precisamos construir laços aninhados. A implementação de laços aninhados é feita da mesma forma com que aninhamos estruturas de decisão, ou seja, basta colocar, como parte do corpo da estrutura de repetição mais externa, uma segunda estrutura de decisão, que controla as operações de repetição que devem ser aninhadas, como na figura 4.3.

Não existem restrições quanto ao tipo de estrutura que aninharemos, isto é, podemos aninhar **while**'s dentro de **for**'s, **do-while**'s dentro de **while**'s, etc. A única exigência feita é de que não exista o entrelaçamento de laços, ou seja, se um laço B está aninhado dentro de um laço A, então ele deve iniciar e terminar dentro do corpo de A, indicando portanto uma relação de continência. A figura 4.3 apresenta, graficamente, como um aninhamento deve ocorrer (a) e como não deve ocorrer (b) (laços entrelaçados), quando se respeita os preceitos da programação estruturadas vistos no capítulo anterior, em especial o da unicidade de caminhos.

EXEMPLO

Um exemplo clássico de aplicação envolvendo o aninhamento de laços é a soma de duas matrizes. Nessa operação devemos ter um laço que percorre as linhas das duas matrizes e outro que percorre suas colunas, como no algoritmo da figura 4.4. Nesse caso, para cada valor da variável *i*, que representa as linhas das matrizes, deve-se alterar o valor da variável *j*, variando-o de forma que a operação de soma seja realizada para todas as colunas de uma determinada linha das matrizes. Assim, o **for** (laço) mais interno é executado um número de vezes correspondente ao número de colunas da matriz para cada execução do **for** (laço)

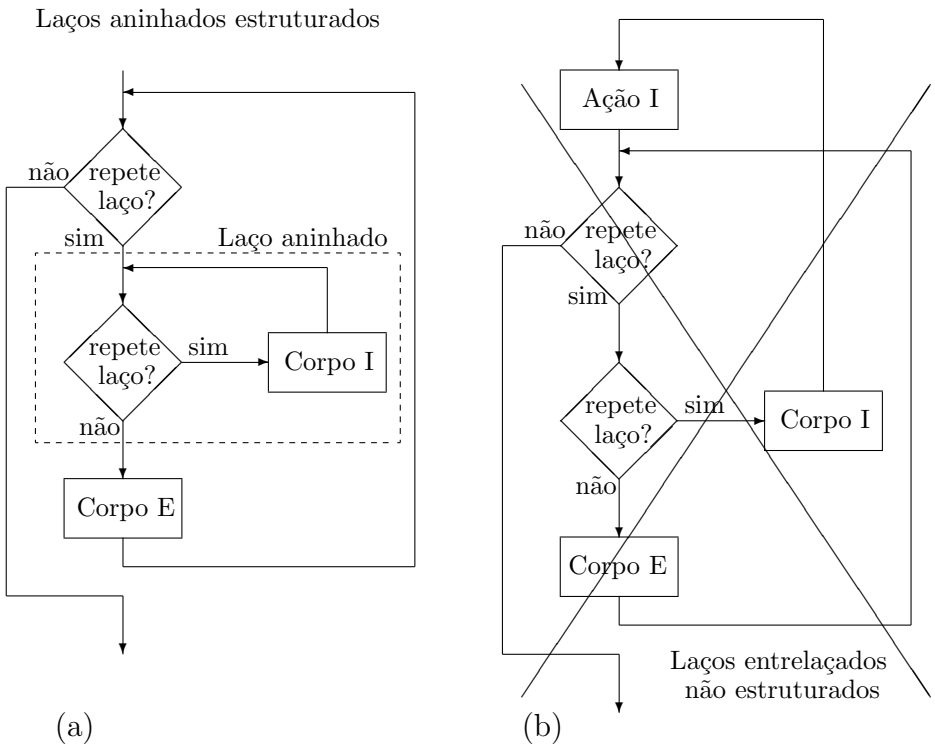

Figura 4.3: Modelos para aninhamento de estruturas de repetição

mais externo. Este, por sua vez, será executado uma quantidade de vezes igual ao número de linhas das matrizes.

Leia os elementos da matriz A
Leia os elementos da matriz B
Para i valendo da primeira linha **até** a última **Faça**
  Para j valendo da primeira coluna **até** a última **Faça**
    $C_{i,j} = A_{i,j} + B_{i,j}$
Escreva o resultado da soma (matriz C)

Figura 4.4: Algoritmo para soma de matrizes

4.7 Laços não estruturados: o comando *goto*

Além das estruturas de repetição apresentadas até agora, muitas linguagens apresentam formas de criação de laços não estruturados, ou seja, laços que não respeitam ao princípio de unicidade de caminhos. Comandos desse tipo (o **goto** do C,

por exemplo) são essenciais em linguagens não estruturadas, tais como Fortran e o Basic. Em C o seu uso é bastante restrito, devendo ser evitado o máximo possível pois a legibilidade de um programa é prejudicada substancialmente ao acrescentar-se *goto*'s. Assim não apresentaremos aqui sua sintaxe, tentando evitar que se aprenda logo algo que não se considera elegante em programação.

Apenas para finalizar essa apresentação de laços não estruturados, o que ocorre na prática com o comando *goto* é forçar com que o fluxo de execução de um programa mude, passando do atual caminho para outro que é identificado através de um rótulo. Comandos desse tipo foram criados nas primeiras linguagens de programação, incluindo-se linguagens de máquina. Eles eram usados então para construir laços equivalentes aos laços estruturados providos nas linguagens modernas. Assim, não faz sentido usá-los, criando códigos complicados, se temos comandos equivalentes que já vêm prontos e são bastante claros na leitura de um programa.

EXERCÍCIOS

1. Escreva algoritmos para resolver os seguintes problemas:

   (a) Ler N números e identificar qual é o maior deles.

   (b) Ler um número, que diga quantos alunos estiveram presentes numa prova e, depois, ler todas as notas, calculando a média da prova.

   (c) Ler um número e, depois, ler N números procurando identificar se o primeiro número lido aparece entre os demais.

   (d) Ler uma série de notas no vestibular, apresentando quantas estiveram na faixa de 0-30, quantas na faixa 31-50, quantas na faixa 51-80 e quantas na faixa 81-100.

   (e) Ler um número qualquer e identificar se ele é primo ou não.

   (f) Ler um par de números positivos diferentes de zero, necessariamente distintos, calculando C_p^n (combinação de n, tomados p a p), em que n é o maior valor lido e p o menor valor lido.

   (g) Ler as coordenadas, no plano, dos vértices de um quadrilátero qualquer, calculando sua área.

2. Escreva programas, em C, que resolvam os problemas do exercício anterior.

Capítulo 5

Tipos de dados compostos

Até o momento temos trabalhado com tipos de dados simples, em que cada variável contém apenas um único valor[1]. Embora possamos programar usando apenas esses tipos de dados, é bastante interessante que tenhamos a possibilidade de agregar valores relacionados entre si em uma única variável. Ao longo desse capítulo veremos o que isso significa e como pode ser realizado.

Numa seção fora deste contexto, apresentaremos também um outro tipo de dados básico em C, que é o tipo endereço. Embora esse tipo de dados seja muito utilizado na construção de estruturas de dados mais complexas, ele também é usado em determinadas formas de se chamar subprogramas, que é o tema do próximo capítulo.

5.1 Tipos estruturados homogêneos

No capítulo 2 examinamos as chamadas cadeias de caracteres. Tais cadeias são conjuntos de caracteres armazenados sequencialmente para formar um nome (ou algo do tipo) e que podem ser acessados referenciando-se uma única variável. Esse tipo de agregação de dados é também muito útil para tipos numéricos em que os vários valores tenham relação semântica entre si. Para entender isso, imagine como poderíamos declarar (e depois manipular facilmente) uma matriz (segundo a definição matemática de matrizes e determinantes) correspondente a um sistema de equações lineares, tal como o sistema a seguir:

$$\begin{aligned} 3x_1 + 2x_2 - 4x_3 &= 10 \\ -x_1 + x_2 + x_3 &= 0 \\ 2x_1 + 4x_2 - x_3 &= 5 \end{aligned}$$

[1] A exceção fica por conta das cadeias de caracteres, em que uma variável contém vários caracteres. Mesmo assim, esses caracteres se referem a um único nome ou frase.

Fazer a declaração das variáveis *a11, a12, a13, a21, a22, a23, a31, a32* e *a33* para armazenar os coeficientes das expressões pode ajudar nesse caso, permitindo o cálculo do determinante e das variáveis x_1, x_2 e x_3. Entretanto, se o sistema de equações for maior do que $3x3$, como por exemplo $100x100$, fica absolutamente impossível declarar individualmente cada coeficiente do sistema (seriam 10000 nesse caso).

A solução a ser adotada nos casos em que valores de mesmo tipo básico apresentam relacionamento entre si é o uso de estruturas de dados homogêneas. Essas estruturas são tipicamente chamadas **vetores** e **matrizes**. Elas são úteis em inúmeras situações, como na representação das matrizes de sistemas de equações, ou de listas com notas de uma disciplina, ou armazenamento de conjuntos de nomes de alunos. Nesse caminho ainda podemos entender as cadeias de caracteres como sendo também estruturas homogêneas que armazenam caracteres relacionados (um nome, por exemplo).

Na elaboração de algoritmos que façam uso de estruturas homogêneas iremos considerá-las como sendo matrizes (segundo sua definição algébrica), com seus elementos sendo acessados sempre de forma individual através de índices que os identifiquem inequivocamente. Uma matriz desse tipo pode ser estruturada com várias dimensões, dependendo da necessidade de organização para a resolução do problema (ou mais formalmente, dependendo de como foi definido o modelo para sua solução). Assim temos matrizes de uma dimensão, que são chamadas de vetores, matrizes de duas dimensões e assim por diante[2].

Podemos agora traçar algumas regras para a definição e manipulação de matrizes:

1. Os elementos de uma matriz são identificados através de índices, que os localizam de forma única dentro da matriz;

2. Não é possível fazer o acesso a vários elementos de uma matriz (uma linha inteira, por exemplo) com uma única operação;

3. Operações sobre elementos de uma matriz são sempre feitas de forma individual (reforçando a regra anterior), através de seus índices.

Um exemplo de manipulação de uma matriz pode ser visto na figura 5.1, supondo que os conteúdos das matrizes *matA* e *matB* foram preenchidos em algum ponto antes da operação de soma das matrizes.

[2]Algebricamente falando, tem-se o caso bastante particular de uma matriz de dimensão zero, que coincide com a definição de uma variável simples!!

Declara matA, matB e matC como matrizes de números reais, de mesma dimensão
⋮
PARA cada linha i de $matC$ FAÇA
PARA cada coluna j de $matC$ FAÇA
$matC_{i,j} \longleftarrow matA_{i,j} + matB_{i,j}$

Figura 5.1: Algoritmo para soma de matrizes usando estruturas homogêneas

5.2 Estruturas homogêneas em C

Vimos até agora qual o propósito de se criar estruturas de dados homogêneas e como elas são manipuladas nos algoritmos e em programas. Para exercitar melhor o uso de tais estruturas é interessante usar exemplos mais concretos, com implementações em linguagem C. Para tanto dividiremos nosso estudo na definição dessas estruturas para o compilador e na manipulação efetiva das mesmas dentro do programa.

Declaração de estruturas homogêneas

Pela descrição do que é uma estrutura homogênea percebe-se que ela é uma reunião de várias posições de memória, que armazenam valores de mesmo tipo, e que tenham alguma forma de relação entre si. Assim, a declaração de uma variável do tipo estrutura homogênea é feita simplesmente através da declaração de uma variável que tenha o tipo básico da estrutura e do número de posições de memória que devem ser reservadas para ela, como nos exemplos a seguir:

```
int umVetor[20], outroVetor[1000], umaMatriz[10][20];
double matrizReais[100][100];
char ooopsUmVetordeCaracteres[20];
```

Nesses exemplos deve-se tomar o cuidado de diferenciar as estruturas numéricas das de caracteres no que diz respeito à quantidade de valores que serão armazenados. Para as cadeias de caracteres é preciso reservar sempre um espaço para o caractere marcador de final da cadeia, enquanto que para as estruturas numéricas isso não é preciso. Assim, as variáveis declaradas no exemplo reservam, respectivamente, posições para 20 inteiros, 1000 inteiros, 2000 inteiros (dispostos em 10 linhas com 20 inteiros/colunas cada), 10000 reais de precisão dupla (dispostos em 100 linhas de 100 colunas cada) e, finalmente, 20 bytes dos quais um deve ser usado para armazenar o marcador de final de cadeia (`'\0'`).

Uso de estruturas homogêneas

Na discussão geral sobre estruturas homogêneas de dados já se antecipou que o acesso a cada elemento da estrutura é feito de forma individualizada. Assim, para se atribuir um valor para uma posição de uma estrutura deve-se fazer uma atribuição indexada precisamente para a posição desejada. Do mesmo modo, para acessar o conteúdo de outra posição também se deve fazer sua indexação.

Assim, para atribuir o valor zero para a primeira posição da estrutura *umVetor* declarada há pouco devemos usar o comando:

```
umVetor[0] = 0;
```

Observe-se que a primeira posição da estrutura é indexada como posição 0. Isso é verdade em C e algumas outras linguagens, sendo que em outras existem regras diferentes para essa indexação. Portanto, para programas em C devemos saber que a primeira posição reservada para a estrutura é a de número 0, sendo as demais sucessivamente indexadas pelos naturais seguintes, até que a última das posições reservadas (a vigésima no caso da variável *umVetor*) terá como índice o tamanho da estrutura menos uma unidade (19 para a variável *umVetor*).

IMPORTANTE!!!
Não existe verificação de limites de estruturas em C. Assim, se você declarar um vetor de dez posições e tentar acessar a décima primeira, seu programa irá ler o conteúdo de uma posição de memória que não pertence ao vetor, provavelmente trazendo lixo (algum valor inconsistente) para a execução.

No caso de estruturas multidimensionais a regra para identificar a posição também é simples. Por exemplo, para a variável *umaMatriz* declarada anteriormente, as posições de memória são reservadas de forma a aparecerem primeiro todas as posições pertencentes à primeira linha, depois todas da segunda linha e assim sucessivamente. Do ponto de vista de índices teríamos a organização lógica apresentada na figura 5.2. Deve ficar claro, entretanto, que a alocação dos espaços disponíveis em memória para armazenar uma estrutura desse tipo não segue exatamente essa disposição espacial.

Na prática toda estrutura multidimensional é armazenada na memória do computador como sendo uma enorme estrutura unidimensional, em que os espaços são reservados dimensão por dimensão. No caso específico da linguagem C, o armazenamento é feito por linhas, ou seja, primeiro reserva-se espaço para todos os elementos da primeira linha, depois para todos os da segunda linha e assim por diante, até a última linha.

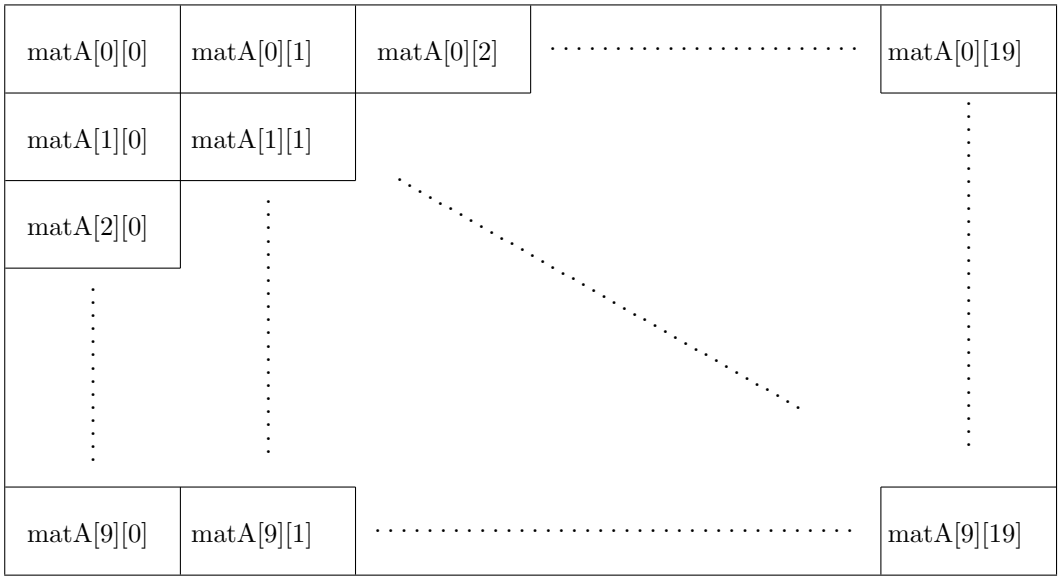

Figura 5.2: Indexação de estrutura bidimensional

Outras linguagens podem usar outras formas de armazenamento, como é o caso do Fortran, que armazena as matrizes por colunas. O conhecimento sobre como uma dada linguagem armazena estruturas multidimensionais é importante para a escrita de programas eficientes, que aproveitem a organização espacial dos dados. De forma simplificada, se a linguagem armazena os elementos da matriz por colunas, então é sempre mais eficiente percorrer seus elementos caminhando por colunas. Percorre-la ao contrário pode levar facilmente a tempos elevados para a busca das informações na matriz.

Feitas essas observações sobre o armazenamento e indexação de estruturas multidimensionais já temos condições de voltar ao algoritmo apresentado na figura 5.1 e construir um programa em C que faça a soma de duas matrizes. Diferente do que foi feito lá, agora trataremos também da leitura dos elementos das matrizes. O programa para tanto é visto na figura 5.3. Nesse código deve ser observado que usamos quatro estruturas de repetição. Uma para ler os dados de *matA*, outra para ler os dados de *matB*, uma terceira para efetuar a soma das matrizes e finalmente um quarto laço para apresentar a soma em *matC*.

Uma solução alternativa poderia fazer a leitura das matrizes em um único laço (como aparece em um comentário no código do programa) e a soma e impressão do resultado também em um único laço. Para tanto deveria-se ter um cuidado adicional na leitura dos dados, de forma a que o usuário não confundisse elementos

```
#include "stdio.h"

main()
{int i, j, matA[10][10], matB[10][10], matC[10][10];

  for (i=0; i<10; i++)      // Para ler a matriz matA
    for (j=0; j<10; j++)
    { printf ("Dê o valor da linha %d e coluna %d de A\n",i,j);
      scanf ("%d", &matA[i][j]);
    }
  for (i=0; i<10; i++)      // Para ler a matriz matB
    for (j=0; j<10; j++)
    { printf ("Dê o valor da linha %d e coluna %d de B\n",i,j);
      scanf ("%d", &matB[i][j]);
    }
// As leituras poderiam ser feitas em um único laço de repetição,
// lembrando-se que se leria um valor de cada matriz a cada vez.

// O mesmo pode ser feito para os laços de soma e impressão
  for (i=0; i<10; i++)
    for (j=0; j<10; j++)
       matC[i][j] = matA[i][j] + matB[i][j]; // Faz a soma
  for (i=0; i<10; i++)
  { for (j=0; j<10; j++)     // imprime resultados de uma linha
       printf (" %d ", matC[i][j]);
    puts(" "); // muda para a próxima linha
  }
}
```

Figura 5.3: Programa em C para a soma de matrizes

de matrizes distintas. Isso seria feito informando o usuário, de forma bastante clara, sobre qual elemento e de qual matriz se deseja ler o valor.

Entretanto, essa não é uma estrutura lógica para a entrada de dados. Normalmente nos preocupamos com uma das matrizes por vez, o que nos leva de volta ao código da figura 5.3. Já para as operações de soma e de impressão dos resultados não existe nenhum cuidado a ser tomado e, podemos até considerar essa junção como uma atitude lógica. Infelizmente, essa otimização é possível apenas em poucas situações.

EXEMPLO

Construa um programa que leia as notas de uma dada prova, calcule sua média e apresente como resultados a média e o número de alunos que tiraram nota maior ou igual a essa média. Considere que o leitura das notas (máximo de 200 notas) termina com a leitura de um valor negativo.

Do enunciado do problema tem-se que o computador deve inicialmente armazenar as notas em alguma estrutura para depois ter como apresentar quantas estavam acima da média, uma vez que essa é calculada após a leitura de todas as notas. Essa estrutura é claramente um vetor de números reais pois admitimos notas em décimos.

O algoritmo (já otimizado) para essa operação é visto na figura 5.4. Observe-se que ao mesmo tempo em que armazenamos as notas já fazemos a sua soma e contamos quantas notas foram lidas, economizando alguns laços de repetição.

Declare um vetor *Notas[200]* para valores reais
Declare variáveis reais *SomaNotas, Media*
Declare variáveis inteiras *k, MaiorIgual*
Faça SomaNotas = 0.0; k = 0; MaiorIgual = 0;
Leia uma nota e guarde em Notas[k]
Enquanto Notas[k] $\geqslant$ 0 **Faça**
    SomaNotas = SomaNotas + Notas[k];
    k = k + 1;
    Leia uma nota e guarde em Notas[k]
Fim enquanto
Media = SomaNotas / (k-1); (lembrar que a última nota lida era negativa)
k = 0;
Enquanto Notas[k] $\geqslant$ 0 **Faça**
    Se Notas[k] $\geqslant$ Media **Então**
        MaiorIgual = MaiorIgual + 1;
    k = k + 1
Fim enquanto
Imprima os valores de Media e MaiorIgual

Figura 5.4: Algoritmo para problema da média das notas

Na figura 5.5 temos o programa correspondente, que dispensa maiores comentários, sobre a utilização de vetores.

```
#include "stdio.h"

main()
{double Notas[200], SomaNotas, Media;
 int i, k, MaiorIgual;  // k guarda o número de notas lidas

   SomaNotas = 0.0;     k = 0;    MaiorIgual = 0;
   scanf("%lf", &Notas[k]); // leitura da primeira nota
   while (Notas[k] >= 0.0)  // laço para ler, contar e totalizar
                            // as notas
   { SomaNotas += Notas[k];
     k++;
     scanf("%lf",&Notas[k]);
   }
   Media = SomaNotas / k; // calcula média após ler nota negativa
// percorre vetor contando notas acima ou iguais a média
   for (i=0; i<k; i++)
      if (Notas[i] >= Media)
          MaiorIgual++;
   printf ("Media = %lf, ", Media);
   printf ("com %d notas iguais ou acima\n", MaiorIgual);
}
```

Figura 5.5: Código C para o programa da média de notas

5.3 Cadeias de caracteres

No capítulo 2 vimos como cadeias de caracteres podem ser manipuladas e para que elas podem ser usadas. Como indicado naquele momento, exemplos mais elaborados seriam deixados para este capítulo. Examinaremos a seguir um exemplo de uso de cadeias de caracteres para identificação de pedaços de uma cadeia de DNA. Faremos essa identificação por duas abordagens distintas, com resultados diferentes em complexidade e eficiência. Antes porém, revisemos as principais funções para tratamento de cadeias de caracteres:

```
strcpy(str1, str2); // copia o conteúdo de str2 sobre o conteúdo
                    // original de str1
```

```
strcat(str1, str2); // copia o conteúdo de str2 no final de str1

strcmp(str1, str2); // compara os conteúdos de str1 e str2,
                    // retornando 0 se forem iguais

strstr(str1, str2); // retorna o endereço da primeira ocorrência de
                    // str2 dentro de str1 (ou nulo se não ocorrer)
```

EXEMPLO

Nesse exemplo apresentamos um programa que lê uma sequência de bases de uma cadeia de DNA (adeninas, guaninas, tiaminas e citosinas, ou A,G,T,C por simplicidade) e procura pela ocorrência de uma determinada subsequência, com ambas as cadeias lidas a partir de teclado. Uma primeira versão é a seguinte:

```
// Versão 1 - sem uso de funções especiais de comparação
#include "stdio.h"

main ( )
{char dna[1000], cadeia[20];
 int i, j, encontrada=0, tamanhoDNA, tamanhoCadeia;

  puts ("Digite a sequência de DNA original");
  scanf ("%s", dna);
  puts ("Digite a cadeia a ser buscada");
  scanf ("%s", cadeia);

// Calcula o tamanho de cada cadeia de caracteres (DNA's)
  tamanhoDNA = strlen (dna);
  tamanhoCadeia = strlen (cadeia);
// Busca ocorrências de 'cadeia' em 'dna'.
// O limite da busca é o tamanho da cadeia do DNA original
// menos o tamanho da cadeia procurada.
//     Porque??
  j = 0;
  for ( i=0; i < (tamanhoDNA - tamanhoCadeia) ; i++)
  { if (dna[i] == cadeia[j]) // encontrei a primeira base (j=0)
       { do
           j++; // vou verificar o resto das bases da cadeia
```

```
          while ((dna[i+j] == cadeia[j]) && (j < tamanhoCadeia));
          if (j == tamanhoCadeia)
              { printf (" Cadeia %s encontrada no DNA\n", cadeia);
                i = tamanhoDNA;
                encontrada = 1;
              }
            else // ainda não encontrei a cadeia
              j = 0;
        }  // final do "if (dna[i] == cadeia[j])"
  }  // final do comando "for"
  if (encontrada == 0)
      printf (" Não foi encontrada a cadeia %s\n", cadeia);
}
```

Uma versão mais prática faz uso das funções de manipulação de cadeias de caracteres. Nela usamos a função *strstr()*, que identifica a primeira ocorrência de uma cadeia dentro de outra cadeia maior. Essa nova versão é apresentada a seguir:

```
// Versao 2 - com uso de funções especiais de comparação
#include "stdio.h"
main ( )
{ char dna[1000], cadeia[20];
  char *inicio; // Marca o endereço inicial de uma cadeia
  int i, j, encontrada=0, tamanhoDNA, tamanhoCadeia;

    puts (" Apresente a sequência de DNA original");
    scanf ("%s", dna);
    puts (" Apresente a cadeia a ser buscada");
    scanf ("%s", cadeia);
    inicio = strstr (dna, cadeia); // compara as cadeias
    if (inicio) // verdade se encontrou a cadeia
        printf (" Cadeia %s encontrada no DNA\n", cadeia);
      else
        printf (" Não foi encontrada a cadeia %s\n", cadeia);
}
```

Nesse exemplo acabamos de examinar duas formas distintas de manipular cadeias de caracteres. Ter a oportunidade de escolha na forma de tratamento

de um problema é sempre interessante. Felizmente é possível verificar que quase sempre existem soluções distintas para o mesmo problema[3]. Isso ocorre primeiro porque o mesmo problema pode ter sua solução modelada de maneiras distintas, e segundo porque um mesmo modelo pode ter algoritmos diferentes para trata-lo. Como então escolher a forma em que faremos a implementação do programa?

Essa escolha depende, em parte, de preferencias pessoais. Como um guia prático podemos fazer essa escolha baseando-se nos dois aspectos a seguir:

1. Clareza e legibilidade

   Aqui se avalia se a solução adotada permite a criação de um programa legível e fácil de se entender. Deve-se destacar que esses são os objetivos primordiais de programação estruturada e devem ser buscados sempre, mesmo que algumas concessões de desempenho tenham que ser feitas.

2. Eficiência

   Nem sempre programas legíveis são os mais eficientes em termos de tempos de execução. Então, em situações em que a velocidade de execução seja uma restrição muito importante, acaba sendo necessário usar-se estruturas de comandos como as que aparecem na primeira versão desse exemplo. Tais estruturas são mais rápidas por fazerem diretamente o que a função *strstr* faria, sem a sobrecarga de processamento que aparece com o tratamento de uma chamada de função.

Para concluir o tratamento de cadeias de caracteres em C é bom relembrar que os tipos caracteres permitem o tratamento de símbolos pelo computador, o que não poderia ser feito diretamente considerando-se apenas tipos numéricos. Além disso sabemos agora que existem várias formas de se tratar cadeias de caracteres e que a escolha entre elas depende das necessidades do programa.

EXERCÍCIO

Escreva os algoritmos e respectivos programas para resolver os seguintes problemas, considerando sempre a leitura[4] pelo teclado e que a indicação do final da entrada de dados, quando não especificada de modo direto, é feita por um par de zeros na forma "00":

[3] Observe que isso é verdadeiro para qualquer problema e não apenas para aqueles que tratem de caracteres.

[4] Observe que para o comando $scanf("\%s", ...)$ o caractere "espaço" indica ao computador o final da cadeia.

1. Ler nomes de pessoas até encontrar um nome começando com a letra Z;
2. Ler uma frase e contar quantas letras aparecem na mesma;
3. Ler uma frase, que pode ter vários espaços em branco entre cada palavra, reescrevendo-a com apenas um espaço em branco entre cada palavra;
4. Ler nomes de animais e trocar todas as ocorrências da letra C pela letra K (maiúsculas e minúsculas);
5. Ler o nome de um objeto e depois procurar se o mesmo aparece numa lista de objetos;
6. Ler uma palavra e verificar se a mesma é um palíndromo (palavras que podem ser lidas de qualquer direção, como "ovo", "arara").

5.4 Tipo endereço: os ponteiros

Um tipo básico muito especial em várias linguagens de programação é aquele destinado à manipulação direta de endereços de memória. Esses tipos são chamados de **ponteiros** ou **apontadores** (do inglês *pointers*) e são largamente utilizados, principalmente quando se trabalha com reserva dinâmica de espaços na memória para os chamados tipos dinâmicos de dados (que é um tópico mais avançado e será abordado a partir do capítulo 8).

Embora tenham uma concepção bastante simples, os tipos ponteiros geram muita confusão no momento de escrever algoritmos e programas. Isso porque eles são, na prática, uma forma de indexação ou redirecionamento, em que ao nos referirmos ao conteúdo de uma variável ponteiro queremos, na realidade, nos referir ao endereço de memória em que está armazenada a variável de nosso real interesse.

Na figura 5.6 apresenta-se o que de fato ocorre na memória, assumindo-se que existe uma variável do tipo ponteiro chamada X, que aponta para um endereço correspondente ao espaço ocupado pela variável A. Assim, ao examinarmos o conteúdo do ponteiro X não encontramos um valor inteiro, real ou caractere. Encontramos sim um endereço de uma posição da memória, que por sua vez conterá um inteiro, real ou caractere (neste caso com valor *Va*).

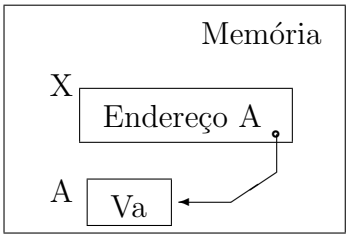

Figura 5.6: Uso de um ponteiro para acessar um endereço na memória

O leitor mais atento deve estar se perguntando agora qual a razão para se complicar aquilo que era simples, ou seja, porque fazermos acessos de forma indireta (ou indexada por endereço) se podíamos fazer esse acesso de forma direta? A resposta para essa pergunta aparece nos vários usos que se pode fazer da manipulação explícita de endereços na memória.

O primeiro deles é a possibilidade de acessarmos diretamente os endereços que armazenam as variáveis do programa, ignorando seus nomes simbólicos. Isso é a base para os procedimentos de passagem de parâmetros por referência que examinaremos no próximo capítulo[5].

Um segundo uso é a alocação dinâmica de memória, em que apenas reservamos espaços na memória quando necessitamos deles. Isso é útil, por exemplo, quando temos que armazenar uma quantidade muito grande (e inicialmente desconhecida) de dados para a execução do programa. Vetores podem ser úteis para armazenar tais dados se forem do mesmo tipo. Porém, como declararmos um vetor que pode ter entre 100 e 1.000.000 de elementos, dependendo da base de dados utilizada? A ideia aqui é lançar mão de estruturas dinâmicas, reservando apenas 100 posições se não precisamos mais do que isso ou qualquer outra quantidade se precisarmos de mais. Isso apenas pode ser obtido com a manipulação direta dos endereços ocupados.

Outros usos incluem o acesso a posições específicas de um vetor ou de uma cadeia de caracteres considerando-se apenas o endereço da primeira posição e o deslocamento entre essa posição e a posição desejada. Na realidade, tanto os *strings* quanto os vetores em C já possuem uma definição implícita de ponteiros para suas primeiras posições.

Assim, vemos que o uso de ponteiros é útil em algumas situações, típicas de problemas de grande porte, e que portanto não devem ser encarados como um problema e sim como uma solução. Deve-se tomar o cuidado, evidentemente, de não confundir o conteúdo da variável ponteiro com o conteúdo da variável para

[5]Esse uso é, na verdade, a razão para a inclusão dessa seção sobre ponteiros neste ponto.

a qual ele aponta. Confusões desse tipo podem, em geral, levar a situações de erro na execução do programa, principalmente se os endereços armazenados na variável ponteiro não pertencerem ao espaço definido para o programa.

5.4.1 Ponteiros em C

O uso de ponteiros em C é relativamente simples. Basta definir, na declaração da variável ponteiro, qual é o tipo base do endereço a ser apontado. Isso significa dizer ao computador que a variável ponteiro estará armazenando um endereço de um inteiro, ou de um real e assim por diante. Implica também que uma variável ponteiro não poderá apontar para um endereço correspondente a um inteiro em um momento e a outro endereço correspondente a um real em outro momento.

A sintaxe de declaração de um ponteiro em C é dada por:

Tipo_Basico *nomePonteiro;

Em que o símbolo * representa o fato de o conteúdo da variável *nomePonteiro* ser um endereço de uma posição da memória que armazena um valor do tipo *Tipo_Basico*.

A manipulação de ponteiros envolve duas situações distintas. Uma em que queremos acessar o conteúdo do próprio ponteiro, ou seja, manipular o endereço armazenado, e outra em que queremos acessar a posição de memória por ele endereçada. Essas situações aparecem nos trechos de programa a seguir, iniciando-se com a atribuição de um endereço ao ponteiro.

```
      ...
int umInteiro, outroInt, *apontaInt, *outroPont;
      // umInteiro e outroInt armazenam valores inteiros
      // enquanto apontaInt e outroPont armazenam endereços de
      // posições na memória que armazenam inteiros
   umInteiro = 10;    // atribui o valor 10 para "umInteiro"
   apontaInt = &umInteiro;  // atribui o endereço de "umInteiro"
                            // para o ponteiro "apontaInt"
      ...
```

O símbolo `&` no último comando indica ao compilador que se quer atribuir, na realidade, o endereço de memória ocupado pela variável *umInteiro* e não o seu conteúdo (10 no exemplo). Após essa atribuição a variável *apontaInt* armazena o

endereço da variável *umInteiro*. Dessa forma a atribuição

```
outroPont = apontaInt;
```

faz com que a variável *outroPont* também armazene o endereço de *umInteiro*.

Por outro lado, a atribuição

```
outroInt = outroPont;
```

provavelmente implicaria em erro de execução (caso não fosse acusado erro de compilação), pela atribuição entre tipos incompatíveis (um endereço atribuído a um inteiro).

Então como poderíamos fazer a variável *outroInt* assumir o conteúdo da variável *umInteiro*, usando o ponteiro *outroPont* para isso? A atribuição a seguir faz isso.

```
outroInt = *outroPont;
```

Nesse caso, ao acrescentar o símbolo *, estamos dizendo ao programa para atribuir o conteúdo da posição endereçada por *outroPont*, que aqui é o endereço da variável *umInteiro*, para a variável *outroInt*.

Ponteiros para operações em estruturas homogêneas

Como apresentado no início desta seção, na linguagem C os ponteiros estão implicitamente associados às posições iniciais de cadeias de caracteres e de vetores. A manipulação dessas estruturas a partir de ponteiros é também bastante simples, como mostram os comandos apresentados a seguir. Neles consideramos inicialmente que tenham sido declaradas as variáveis *i*, inteira, e *umVetor*, como sendo um vetor de 100 inteiros. O comando `for` atribui 0 para cada uma das posições do vetor.

```
for (i=0; i<100; i++)
   *(vetor + i) = 0; // atribui 0 para todas as posições do vetor
```

Como se pode ver, para acessar uma determinada posição do vetor basta passar o endereço da primeira posição, que seu próprio nome, somado com o índice da posição que se quer acessar. Isso fica mais evidente no comando a seguir, que copia o conteúdo de uma posição do vetor, a décima no exemplo, para uma variável inteira.

```
i = *(vetor + 9); // atribui o conteúdo da décima posição
                  // do vetor para a variável i
```

Assumindo agora que tenha sido declarada uma cadeia de caracteres *nome*, de forma convencional (`char nome[30];`) e outra cadeia de caracteres chamada *outroNome*, usando ponteiros (`char *outroNome;`), temos os comandos exemplificados a seguir.

```
strcpy(nome,"UmNomeArretado");
outroNome = nome;  // atribui o endereço da primeira posição
                   // de 'nome' para o ponteiro outroNome
printf("O nome é... %s\n", outroNome); // imprime UmNomeArretado
if (*(outroNome+2) == 'N')
    puts("Terceira letra é N");
```

Vale observar aqui que no caso de *strings* não é necessário indicar que queremos imprimir o conteúdo do endereço armazenado no ponteiro de caracteres. Essa é uma peculiaridade de cadeias de caracteres, em que o endereço da primeira posição é assumido como sendo o início da cadeia e a sua última posição é marcada pelo caractere `'\0'`, como visto anteriormente.

EXERCÍCIOS

Escreva os algoritmos e respectivos programas para resolver:

1. Cálculo do determinante de matrizes 3x3.
2. Implementação do jogo da velha.
3. Cálculo da média global dos alunos de uma sala, conhecendo-se as notas dos mesmos em cada uma das disciplinas do curso (por exemplo matemática, português, história, geografia, física, biologia e química). Assuma um máximo de 50 alunos na turma.
4. Para os alunos da sala anterior calcule a média da sala em cada disciplina e conte quantos alunos estão acima da média em pelo menos quatro disciplinas.
5. Cálculo do centro de massa de um conjunto de pontos (x,y) num plano, dado o número de pontos e suas coordenadas.
6. Identificação do ponto que esteja mais próximo do centro de massa calculado no exercício anterior.

Capítulo 6

Subprogramas

Até esse momento temos considerado nossos programas como formados de um só corpo, no qual o fluxo de execução segue linearmente, do início ao fim do programa, com alguns poucos desvios causados pelas estruturas de decisão e de repetição. Ocorre muitas vezes que essa estrutura extremamente monolítica se torna exageradamente grande. Por exemplo, um programa para fazer a inversão de matrizes não pode ser escrito em poucas linhas (na realidade, dependendo do método usado, precisaremos bem mais do que cem linhas para tanto). Tente então imaginar como seria escrever um programa, da forma como estamos fazendo, em duas ou três mil linhas. É quase impossível ler esse programa, depois de pronto, para tentar descobrir o que é que ele faz e como é que ele atinge seus objetivos.

Para resolver esse problema usa-se a técnica de particionar o programa em partes menores (lembram-se da técnica de modelar o problema em sub-problemas menores e de resolução mais simples?), as quais seriam executadas quando fossem necessárias. Isso é obtido com o uso de chamadas de subprogramas[1] pelo programa principal.

A ideia utilizada no conceito de subprogramas é equivalente ao processo de delegar tarefas a ajudantes. Um exemplo disso envolveria fazer um bolo e deixar alguém encarregado de bater a massa. Assim, ficaríamos encarregados de colocar numa tigela todos os ingredientes da massa. Ao completar essa operação bastaria chamar a pessoa encarregada de bater a massa, passar a ela a tigela com os ingredientes e esperar pela entrega da massa, já batida.

No computador é exatamente isso que ocorre no particionamento do problema por meio de subprogramas. O programa tem alguém, o tal do subprograma, especializado na execução de uma determinada tarefa, que é chamado no momento em que a execução da tarefa é necessária. O programa fornece ao subprograma

[1]Subprogramas são chamados de funções, procedimentos, sub-rotinas, tarefas, métodos, etc., conforme a linguagem usada.

todas as informações exigidas para a sua execução, deixando-o então trabalhar até que a tarefa seja concluída. Nesse momento o subprograma devolve, junto com os resultados de sua operação, o controle da execução ao programa, que iria então concluir sua atividade.

Na realidade, a ideia por trás de subprogramas não é simplesmente separar o programa em pequenas partes isoladas e sequenciais. O objetivo é, isso sim, identificar partes do programa que tenham que ser executadas muitas vezes durante a vida do mesmo, isto é, pontos distintos do programa mas que executem a mesma tarefa. Um exemplo interessante disso é um programa que faça o controle de uma biblioteca. Nele, toda vez que quisermos reservar um livro, marcar livros como emprestados ou desmarca-los após a devolução, teremos que ler o nome do livro e achar onde é que está a sua ficha dentro do computador. Pois bem, se separarmos as rotinas de leitura de nomes de livros e a de busca por fichas do restante do programa, poderemos facilmente fazer com que essas rotinas sejam ativadas toda vez que forem necessárias, mesmo que sejam chamadas de pontos diferentes do programa, como marcação de empréstimos ou verificação de reserva. Isso ocorre ainda com a vantagem de aparecerem fisicamente apenas uma vez em todo o programa.

Uma forma bastante simples, porém inexata, de imaginar como o computador usa subprogramas é imaginar que em cada ponto que um subprograma é chamado, o computador coloca no lugar dessa chamada todo o código correspondente ao mesmo, como apresentado na figura 6.1. Esse raciocínio facilita a visualização do processo de utilização de subprogramas, porém não descreve o que de fato ocorre no computador.

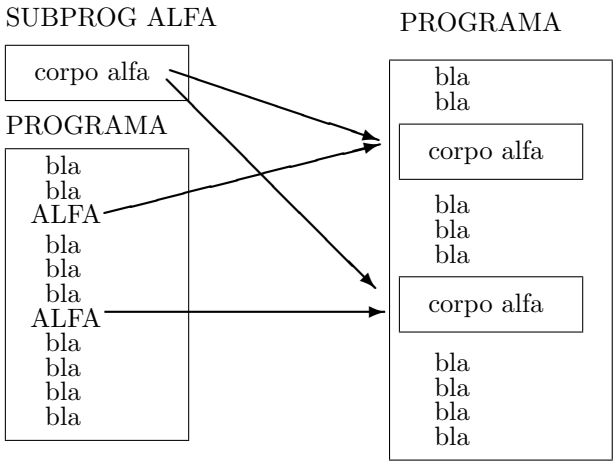

Figura 6.1: Visão conceitual da execução de subprogramas

O processo real, visto na figura 6.2, consiste em fazer com que o fluxo de

computação passe do ponto em que ocorreu a chamada para o ponto em que está localizado fisicamente o subprograma, executando-o e retornando depois ao ponto em que se deu a chamada. O controle do fluxo de execução aqui descrito é feito por meio da Unidade de Controle do processador, mais precisamente pelo valor do registrador denominado Contador de Programa (PC).

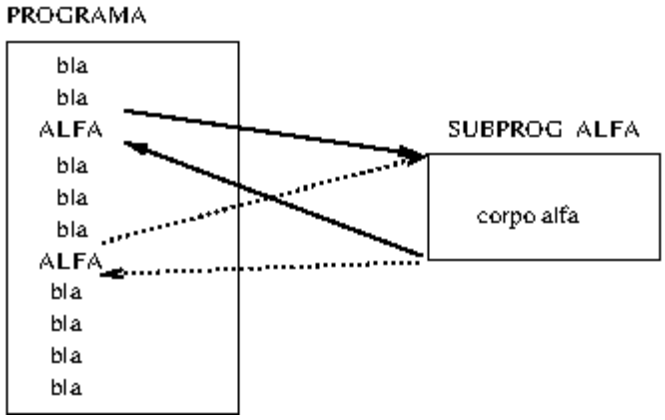

Figura 6.2: Visão de fluxos da execução de subprogramas

Como podemos ver, o funcionamento real do programa durante uma chamada de subprograma proporciona uma grande economia em termos de espaço que o programa vai ocupar na memória. O problema é que essa economia vem com prejuízos ligados à velocidade de execução do mesmo, pois cada vez que o programa é desviado de seu curso original temos que salvar conteúdos de registradores internos da CPU e restaurar valores que sejam úteis para o novo trecho do programa em execução. Isso nos leva a usar o recurso de chamadas de procedimentos e funções de forma criteriosa, ou seja, apenas podemos lançar mão deles quando o seu uso implicar em melhorias no estilo de programação.

A decisão sobre quando usa-los deve ser feita pelo programador no momento da especificação do algoritmo. Infelizmente, os critérios para fazer essa tomada de decisão não são precisos, ficando na maioria das vezes restrito ao modo como o programador trabalha e sua experiência pessoal sobre tais circunstâncias.

Existem vários detalhes que devem ser observados na implementação real de subprogramas, mas os deixaremos para depois por serem dependentes da linguagem em uso. Aqui nos preocuparemos apenas com aspectos gerais do uso de subprogramas, de forma a estabelecer uma metodologia para sua aplicação. Para isso definiremos uma sintaxe algorítmica para subprogramas, que segue o seguinte padrão:

```
SUBPROG  nome_subprograma ( lista_de_parâmetros )
DECLARA  variáveis_usadas_no_subprograma
   COMANDOS DO SUBPROGRAMA
   (terminando, se necessário, com comando "RETORNE valor")
FIM SUBPROG
```

Este subprograma seria ativado, dentro do programa, por um comando do tipo:

```
    nome_subprograma ( lista_de_valores_ou_variáveis );
ou
    variável = nome_subprograma ( lista_de_valores_ou_variáveis );
```

Sobre as lista de valores/variáveis e de parâmetros deve-se observar que:

1. São dados que o programa deve passar ao subprograma como informações necessárias ao seu funcionamento;
2. A ordem em que cada dado aparece deve ser a mesma tanto no subprograma quanto no ponto de sua ativação;
3. Deve haver correspondência entre a quantidade e os tipos das variáveis da chamada para com os parâmetros do subprograma;
4. Os nomes simbólicos das variáveis na lista de parâmetros podem ser completamente distintos daqueles na lista de variáveis;
5. Podem não existir se o subprograma não necessitar de parâmetros.

Examinaremos agora com mais cuidado a existência desses parâmetros e a validade dos mesmos ao longo do programa.

6.1 Parâmetros e escopo de variáveis

Da discussão anterior tornou-se clara a existência de parâmetros, que nada mais são do que argumentos caracterizando o conjunto de informações necessárias para a execução correta do subprograma. Em alguns casos eles também caracterizam resultados produzidos com a execução do subprograma, principalmente nos subprogramas que não retornam respostas de forma explícita a quem os ativou. Esse aspecto, entretanto, será tratado mais adiante, no momento adequado para isso.

Escopo

A necessidade do uso de parâmetros surge apenas porque nem toda posição de memória é acessível (ou está visível) em todo ponto da execução do programa (ou subprogramas). A definição de acessibilidade (ou visibilidade) das posições de memória usadas por um programa (e portanto suas variáveis) é feita pela definição de **escopo** de variáveis. Todas as linguagens apresentam a possibilidade de se trabalhar com escopos diferentes segundo o nível de modularização (blocos e subprogramas) está sendo usado.

O problema é que ao se criarem escopos diferentes estamos também criando a necessidade de se estabelecer pontos de visibilidade dos vários endereços de memória. Mais do que isso, ao ocultar um determinado endereço (contendo um valor usado no programa) em parte da execução do programa, como um subprograma por ele ativado, estamos impedindo que aquela informação seja usada diretamente. Isso implica, finalmente, na necessidade de passagem de parâmetros entre programa e subprograma.

Surge então a questão de porque não termos apenas um escopo para as variáveis, evitando assim a passagem de parâmetros. Existem duas justificativas importantes para isso.

A primeira delas é que ao definirmos nomes simbólicos com validade em apenas um trecho do programa se torna possível reutilizar um dado nome, para outras finalidades, em outros trechos do programa. Isso significa que não precisamos criar nomes diferentes para todas as posições de memória que serão utilizadas durante a execução do programa. Essa é, como veremos com o tempo, uma vantagem enorme em relação ao uso de apenas variáveis com escopo global (com nomes válidos em todo o programa).

Uma segunda justificativa está relacionada à economia de posições de memória obtida com a definição de variáveis apenas quando forem necessárias. A economia surge porque teríamos variáveis seriam visíveis apenas em pequenos trechos do programa, portanto locais a esses trechos. Tais variáveis existiriam (ocupariam espaço) na memória por pequenos intervalos de tempo. Esses espaços poderiam então ser ocupados por outras variáveis, locais a outros trechos do programa, em outros instantes. Já para o caso de variáveis de escopo global temos que o espaço reservado a elas não pode ser liberado em momento algum da execução do programa, resultando em algum desperdício de espaço se a definição de variáveis com esse perfil não for feita com cuidado.

Vemos portanto que é vantajosa a existência de escopos distintos para a manipulação de variáveis. O problema é que caso um subprograma esteja executando em um escopo, ele não terá acesso a variáveis de outros escopos. Para que ele possa usar dados contidos em outros escopos o programa deverá recebe-los na forma de parâmetros, que são transportados entre quem ativa um subprograma

e o próprio subprograma.

O transporte das informações entre escopos distintos pode ser feita de várias formas. Elas estão associadas com as maneiras de reservar e acessar posições na memória ou, de outro modo, associadas com a duração de sua existência. Assim, temos basicamente três tipos de existência de dados, que são na forma de parâmetros, de variáveis locais e de variáveis globais.

O entendimento das relações entre parâmetros, escopo e posições de memória ocupadas pelo programa é fundamental para que se entenda o funcionamento de subprogramas e como eles podem ser usados de forma coerente. Assim, algumas definições são necessárias:

- Variável global:

  Trata-se de uma variável cujo endereço na memória é acessível por qualquer parte do programa, usando-se o nome simbólico que foi declarado como sendo global. Isso implica em que os endereços por elas ocupados existem durante toda a execução do programa. A sua declaração deve ocorrer no início da execução, para permitir o seu acesso a qualquer instante. Dependendo da linguagem podem existir conflitos quando se declara alguma variável local com o mesmo nome da variável global, fazendo com que a declaração local oculte a global naquele escopo.

- Variável local:

  Trata-se de uma variável cujo endereço de memória é acessível apenas no trecho para o qual foi declarada, sendo que seu nome simbólico pode ser usado apenas durante esse trecho. Normalmente são declaradas, e portanto existem, dentro de subprogramas. O espaço por elas ocupado na memória é liberado (para outros usos) no momento em que o fluxo de execução deixa o trecho em que elas são visíveis (seu escopo).

- Parâmetros:

  São valores passados entre um programa (ou subprograma) e o subprograma por ele ativado. Isso implica na necessidade entre o mapeamento entre os nomes simbólicos de quem ativou o subprograma e nomes simbólicos declarados no subprograma. Esse mapeamento pode ser feito com a cópia de conteúdos entre endereços diferentes na memória ou do próprio endereço representado pelo parâmetro, com resultados evidentemente diferentes em cada caso. Veremos adiante como diferentes formas de passagem de parâmetros causam resultados totalmente diferentes em um programa.

Resumindo, temos um conjunto de variáveis que são declaradas no início da execução do programa e podem ser vistas (ter seu valor lido ou alterado) a partir

de qualquer ponto do programa. Temos outro conjunto que pode ser declarado a qualquer momento e existe apenas dentro do corpo estrutural (um trecho de programa ou subprograma) em que foi criado. Quando precisamos usar essas variáveis dentro de subprogramas em que elas não sejam visíveis temos que passá-las como parâmetros, o que pode ser feito pela cópia da informação ou do próprio endereço da informação, como veremos a seguir.

6.1.1 Passagem de parâmetros

Quando definimos os parâmetros passados para um subprograma pode ocorrer duas situações distintas. Numa primeira situação queremos que o subprograma faça uso do parâmetro sem modificá-lo durante sua execução. Na outra queremos que algum parâmetro seja alterado e que essa alteração seja percebida pelo programa que ativou o subprograma. A figura 6.3 ilustra ambas as situações para um algoritmo de cálculo de médias. A partir dela temos então as seguintes diferenças entre as duas formas de passagem de parâmetros:

```
SUBPROG media (x, y: reais)
  DECLARA media: real;
  FACA  media = (x+y)/2;
  RETORNE valor de media
FIM SUBPROG

PROG
  DECLARA a, b, c: reais
  LEIA a, b
  c = media(COPIA a, COPIA b)
  ESCREVA c
FIM PROG
```

(a)

```
SUBPROG media (m, x, y: reais)
   FACA m = (x+y)/2;
 FIM SUBPROG

PROG
  DECLARA a, b, c: reais
  LEIA a, b
  media (c, COPIA a, COPIA b)
  ESCREVA c
FIM PROG
```

(b)

Figura 6.3: Parâmetros em subprogramas: (a) Com passagem de cópias. (b) Com passagem de endereço e cópias

- Quando passamos cópias temos que retornar o resultado de forma explícita ao programa, o que não ocorre se passamos o endereço de quem receberia a resposta;
- Temos que deixar explícito quais parâmetros são cópias e quais são endereços;

- Variáveis correspondentes aos valores de resposta precisam ser declaradas explicitamente se os parâmetros forem apenas cópias, enquanto correspondem ao próprio parâmetro se for seu endereço.

Tentemos agora enxergar essas diferenças com um exemplo. Imagine um subprograma que troque os valores de duas variáveis. Na figura 6.4 temos uma versão em que as variáveis são passadas como cópias das variáveis originais. Assim, ao executar o programa imprimiremos (erradamente) os valores 0 e 1, nessa ordem. Isso ocorre pois as variáveis alteradas pelo subprograma eram apenas cópias das variáveis originais, as quais não foram alteradas portanto.

Para fazer de fato a troca, e preservá-la ao retornar para o programa que ativou o subprograma, é possível usarmos duas técnicas diferentes. A primeira, altamente desaconselhável, seria usarmos apenas variáveis globais, mas isso limitaria o subprograma a trabalhar sempre sobre o mesmo conjunto de variáveis, independente do trecho do programa que o ativasse. A segunda técnica usa parâmetros que sejam o próprio endereço da variável original, como aparece na figura 6.5. Nesse caso a impressão resultará em 1 e 0, nessa ordem, que é o que se esperava desde o início.

```
SUBPROG troca (x, y : reais)
DECLARA temp : real;
   temp = x;
   x = y;
   y = temp; // troca o conteúdo das variáveis locais
FIM SUBPROG

PROG
DECLARA x, y : reais
   FACA x=0, y=1
   troca (COPIA x, COPIA y)
   IMPRIME x, y
FIM PROG
```

Figura 6.4: Subprograma para troca de variáveis. Versão incorreta.

A figura 6.6 mostra o que ocorre na memória nas duas situações. É importante perceber que no caso de parâmetros passados por cópia são alocadas novas posições na memória, que recebem cópias dos valores das variáveis iniciais, como indicado no quadro referente ao passo $p2$. São essas as posições de memória manipuladas durante a execução do subprograma.

Isso ocorre ao longo dos passos $p3$ (atribui o valor da variável x para $temp$), $p4$

```
SUBPROG troca (END a, END b : reais)
DECLARA temp : real;
   temp = CONT a;  // temp recebe o conteúdo da posição A
   POS a = CONT b; // posição A recebe o conteúdo da posição B
   POS b = temp;  // posição B recebe o conteúdo de temp
FIM SUBPROG

PROG
DECLARA x, y : reais
   FACA x=0, y=1
   troca (END x, END y)
   IMPRIME x, y
FIM PROG
```

Figura 6.5: Subprograma para troca de variáveis. Versão correta.

(passa o valor de y para a variável x) e $p5$ (faz y receber o antigo valor de x, que estava armazenado em *temp*). Ao término dessa execução os espaços alocados para as variáveis locais são sumariamente devolvidos ao sistema e seus valores deixam de existir (passo $p6$).

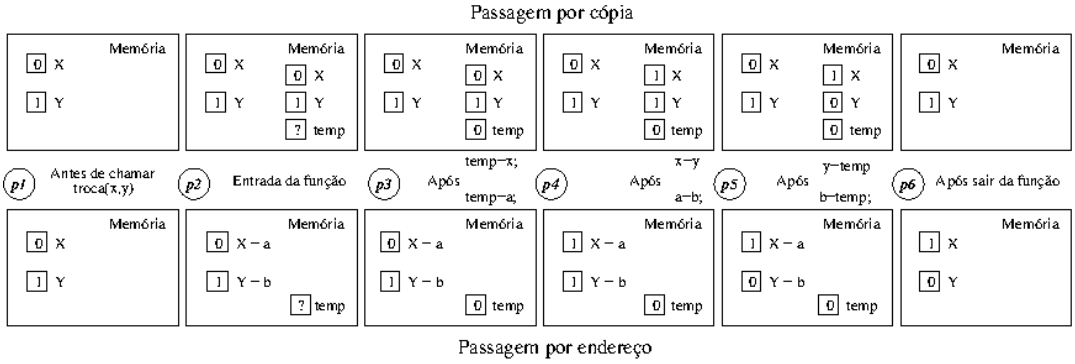

Figura 6.6: Estado da memória nas diferentes formas de passagem de parâmetros

Já para os parâmetros passados por endereços não ocorre a alocação de novos espaços na memória (passos $p1$ e $p2$). Na realidade os nomes simbólicos da lista de parâmetros são associados aos endereços das variáveis originais (passo $p2$). Portanto, as alterações feitas pelo subprograma sobre tais variáveis ocorrem, de fato, nas variáveis originais (passos $p3$, $p4$ e $p5$, que fazem a de troca de conteúdos), embora no subprograma sejam usados nomes simbólicos diferentes daqueles das variáveis originais. Ao final da execução (passo $p6$) as variáveis originais acabam mantendo os valores que assumiram durante a execução do subprograma.

6.1.2 Cuidados com uso de parâmetros

Existem alguns cuidados que devem ser tomados no momento de se fazer a passagem de parâmetros para um subprograma. Esses cuidados estão ligados com a consistência que se deve dar aos conteúdos das variáveis globais e locais. Assim, poderíamos listar as seguintes condições para o uso de variáveis dentro de um subprograma:

1. Deve-se evitar a atribuição, dentro de um subprograma, de valores para variáveis que não sejam locais ao mesmo;

2. Devemos usar, preferencialmente, passagem de parâmetros por cópia de seu valor quando não for necessária a alteração de seu conteúdo;

3. Caso seja necessário alterar-se o conteúdo de uma variável não local, é preferível que se faça pela atribuição para um parâmetro passado por endereço que mapeie essa variável, o que resulta, na prática, em fazer a atribuição para a variável não local desejada;

4. Apenas fazemos passagem de parâmetros pelo endereço da variável para aquelas variáveis que tiverem que ter seus conteúdos alterados dentro do subprograma;

5. O uso criterioso de variáveis globais e locais propicia economia de espaço de memória e facilita a denominação de variáveis ao longo do programa;

6. Deve-se lembrar que o uso de nomes de variáveis com declaração global e local implica em ser válido dentro de um subprograma o nome declarado mais recentemente, isto é, se dentro de um subprograma *SubProgA* tivermos uma variável local *var1*, que também foi declarada como sendo global, então toda referência à variável *var1* dentro de *SubProgA* é relacionada com a variável local, declarada dentro dele.

7. Por fim, apesar dos perigos envolvidos com o uso de subprogramas, isto é, alterações (ou falta de alterações) não previstas de variáveis, o resultado do uso desse recurso de programação é extremamente positivo, reduzindo o tamanho dos códigos fontes dos programas, melhorando sua legibilidade e a facilidade para a sua escrita e correção.

6.1.3 Outras formas de passagem de parâmetros

A passagem de parâmetros por cópia e por referência são as mais comuns e usadas. Porém, não são as únicas formas de passagem de parâmetros. Algumas linguagens

habilitam a passagem de parâmetros por outros mecanismos, com resultados e capacidades diferentes daquelas vistas até agora.

O Fortran habilita a passagem **por resultado**, em que um ou mais argumentos são passados para o subprograma e receberão, no término da execução do subprograma, valores nele definidos. Isso é diferente tanto da passagem por referência (não se passa o endereço do argumento) e de valor de retorno do subprograma (não se atribui esse resultado usando atribuição).

Já a linguagem Ada proporciona passagem por **valor**, por **resultado** e por **valor-resultado**, indicando explicitamente quando um parâmetro é passado por valor (IN), quando é de resultado (OUT) e quando é de valor-resultado ($INOUT$). Essas formas, entretanto, são ainda muito parecidas com as passagem por valor e por referência. A diferença aqui é que na passagem **por resultado** o conteúdo da variável que receberá o resultado é alterado apenas no final da execução do subprograma e não em qualquer atribuição, como ocorre na passagem por referência.

Uma última forma de passagem de parâmetros é a **passagem por nome**. Aqui, os argumentos passados para o subprograma substituem, nominalmente, os parâmetros lá utilizados. Isso funciona aproximadamente como a ideia de **macros** presente em várias linguagens (como o C), embora sua existência como forma de passagem de parâmetros esteja restrita à família de linguagens originada com o Algol. Na passagem por nome ao passarmos um argumento estamos passando uma referência nominal e não um valor ou o endereço do argumento. Isso significa que se algum argumento for alterado no subprograma, sua referência também será alterada. O exemplo apresentado na figura 6.7, escrito em uma notação próxima do C, ilustra o funcionamento da passagem por nome.

```
Function PassBYName (int x, j)
{// Aqui 'm' assume a posição de 'x' e 'c[m]' a posição de 'j'
   x = x+1;      // Aqui faz m=m+1, resultando em m=2
   j = j+2;      // Faz c[m]=c[m]+2. Como m=2, então faz c[2]=4
}                // e não c[1]=3, como na passagem por referência

Function Main()
{int m, c[20];
   c[1]=1; c[2]=2; c[3]=3; ...
   m = 1;
   PassBYName (m, c[m]);
   printf ("%d, %d, %d, %d, ...", m, c[1], c[2], c[3],...);
    // Imprime 2, 1, 4, 3, ...
}
```

Figura 6.7: Passagem de parâmetros por nome.

6.2 Parâmetros e escopo de variáveis em C

Os problemas tratados até o início da página anterior são genéricos, não referindo-se a alguma linguagem em particular. Ao passar dos conceitos algorítmicos para os de implementação em uma linguagem computacional qualquer, temos que estabelecer algumas restrições que viabilizem o trabalho do compilador. Para a linguagem C temos que as passagens de parâmetros e regras de escopo são relativamente simples. Porém, antes de examina-las é preciso apresentar a sintaxe geral de uma função em C, o que é visto na figura 6.8.

```
TipoFunção  NomeFunção ( Lista_de_parâmetros )
{ variáveis internas da função

     Corpo da função

}
```

Figura 6.8: Sintaxe geral de uma função em C

Dessa sintaxe devem ser destacados os seguintes aspectos:

- Todo subprograma em C é uma função, incluindo-se a função *main*, com a qual se inicia a execução do programa;
- `'TipoFunção'` se refere ao tipo do valor retornado após a execução da função, podendo ser *int, float, char, double* ou qualquer outro tipo declarado para o programa, incluindo-se o tipo *void* quando não se retorna valor de forma explícita;
- `'NomeFunção'` é o nome simbólico pelo qual outros trechos do programa poderão ativar (e fazer uso) da função;
- `'Lista_de_parâmetros'` define para a função quais parâmetros ela estará recebendo de quem a ativou, bem como os tipos desses parâmetros, podendo ser vazia caso não se necessite de parâmetros, como é o caso da função *rand()*, que gera números aleatórios;
- `'variáveis internas da função'` são variáveis declaradas internamente à função, sendo portanto de escopo local a ela;
- `'Corpo da função'` corresponde ao código efetivo da função, contendo todas as ações necessárias para que se realize a sua tarefa. Se houver retorno explícito de um valor para quem ativou a função, então é preciso que exista

no corpo da função pelo menos um comando *return (valor)* para fazer esse retorno.

Para a chamada de funções a sintaxe também é simples, consistindo na ocorrência do nome da função, com seus parâmetros, no lado direito de uma atribuição ou como sendo o comando completo, caso não se retorne valor explicitamente. A seguir temos alguns exemplos de chamadas de funções.

```
y = rand();  // chama a função rand e atribui a resposta para y
puts(" Uso da função puts");  // chama puts sem uso de resposta
alfa = func_beta(tg(x), y, 2) + sin(x) / cos(y); // chama as
      // funções tg, sin, cos e func_beta (com argumento tg(x))
```

O que se deve observar aqui é que os parâmetros devem sempre vir entre parênteses, podendo ser variáveis, constantes ou mesmo valores retornados por outras funções. Caso não se necessite de parâmetros é preciso fazer a chamada colocando-se o nome da função seguido de um abre-fecha parênteses sem nada dentro. Um exemplo mais bem cuidado de funções será apresentado após o entendimento dos processos de passagem de parâmetros e de escopo.

Escopo em C

As regras de escopo em C são bastante simples, admitindo três níveis de básicos, que são o global, o de função e o de bloco estrutural[2]. Os aninhamentos de escopo ocorrem estruturadamente, na ordem *global* → *função* → *bloco*, sendo que as regras para definir o escopo de uma variável são as seguintes:

1. Variáveis globais são aquelas declaradas fora do corpo das funções. São visíveis em qualquer parte do programa, sendo possivelmente redefinidas por declarações locais às funções ou blocos estruturais.

2. Variáveis locais às funções são aquelas declaradas no início da função e, quando for o caso, como algum dos parâmetros passados para a função. Sua visibilidade se restringe ao corpo da função, podendo ser redefinidas por eventuais declarações internas aos blocos estruturais.

3. Variáveis locais aos blocos estruturais são aquelas declaradas no início de um bloco estrutural, tendo sua visibilidade restrita ao próprio bloco.

[2]Um bloco estrutural em C é qualquer conjunto de comandos cercado por um abre-fecha chaves { ... }

4. Um nome simbólico definido como global e local assume sempre a declaração mais interna, ou seja, a declaração mais local, retornando para o contexto da declaração mais externa ao término da função ou bloco estrutural.

Parâmetros em C

A passagem de parâmetros em C é sempre feita por cópia da variável, quando é denominada como **passagem por valor**. A passagem por referência pode ser simulada pela passagem (por valor) do endereço da variável. Deve-se observar que a passagem do endereço, no contexto do C, é diferente do que ocorre em linguagens com passagem por referência. Em outras linguagens a manipulação da variável passada por referência é idêntica à manipulação de qualquer outra variável (local, global ou passada por valor), ou seja, se a variável for do tipo *float* é possível usar operadores de números reais sem qualquer cuidado adicional. Em C uma variável passada como endereço tem que ser manipulada explicitamente por meio de ponteiros (eis aqui a razão para termos introduzido o tipo ponteiro na seção 5.4) e não como do tipo original da variável.

Apesar de serem teoricamente equivalentes em C, trataremos esses dois tipos de argumentos (valores e endereços) como sendo formas distintas de passagem de parâmetros. Dessa forma, podemos nos conceder o direito de uma liberdade poética e chamar de "passagem por referência" quando usarmos endereços como argumentos passados para a função (perdão aos pesquisadores em teoria de linguagens de programação). A sintaxe envolvida na passagem e uso de parâmetro, tanto como valor como por referência, é vista a seguir.

Passagem por valor – na chamada da função os parâmetros devem ser valores representados por constantes, variáveis, ou ainda expressões (envolvendo ou não outras funções), como em:

```
x = func_name (var1, var2, ...);
```

Na definição da função os parâmetros devem ser declarados de forma simples, como sendo variáveis do mesmo tipo das variáveis originais, como em:

```
func_type func_name (type1 copy1, type2 copy2, ...)
```

Passagem por referência – na passagem por referência, como tratamos de endereços das variáveis, é preciso fazer uso dos ponteiros examinados no capítulo anterior. Assim, na chamada da função devem ser passados os endereços das variáveis que passaremos por referência, enquanto que na definição da função os parâmetros devem ser declarados como sendo variáveis que apontam para esses

endereços. A sintaxe da chamada, para uma função com apenas um parâmetro, é vista a seguir:

```
x = func_name (&var1);
```

Da mesma forma, a definição dessa função é dada por:

```
func_type func_name (type1 *metavar1)
```

Para acessar o conteúdo da variável passada por referência dentro da função é necessário tratar corretamente os ponteiros. Os comandos a seguir mostram respectivamente como ler e atribuir um valor de/para `var/metavar`:

```
var_x = *metavar;  // var_x recebe o valor armazenado na posição
                   // apontada por metavar

*metavar = 10;     // o valor 10 é atribuído para a posição na
                   // memória apontada por var_x
```

6.2.1 Definição de protótipos

Como C é uma linguagem bastante aberta e baseada na composição de várias (até mesmo milhares) funções, e cada função consegue visualizar (acessar, na realidade) apenas as funções previamente analisadas pelo compilador, é preciso criar mecanismos para aumentar essa visibilidade. Isso é obtido com a definição de **protótipos** das funções. Um protótipo de uma função nada mais é do que a declaração do nome e tipo da função, acompanhado de quantos, e de que tipos, são os parâmetros por ela utilizados.

Isso é feito nos arquivos incluídos nas linhas `#include <biblioteca.h>`, por exemplo. Nesses arquivos temos apenas definições de constantes e protótipos para as funções de uma dada categoria, como as funções matemáticas dentro de **math.h**. Neles não estão os códigos dessas funções.

A definição dos protótipos ocorre usualmente no início do código do programa, junto com a definição das variáveis globais. Um protótipo apresenta apenas a definição do tipo, nome e parâmetros da função, omitindo o seu corpo, que pode então aparecer em qualquer momento do código, na forma definida na figura 6.8. A declaração do protótipo é feita da seguinte forma:

```
TipoFuncao nomeFuncao (TipoParam1, TipoParam2, ..., TipoParamN);
```

O uso de protótipos não é obrigatório, porém facilita muito o processo de compilação e de arranjo da ordem das funções no código. Essa facilitação ocorre porque ao declararmos os protótipos podemos fazer chamadas dessas funções em

qualquer ponto do programa, mesmo que seus códigos ainda não tenham sido compilados. Vale lembrar que a quantidade e tipo de parâmetros de uma função ainda não compilada é desconhecido pelo compilador, gerando então erros de compilação.

> Observe-se que para gerar o código executável do programa teremos que compilar essas funções em algum momento, mas com os protótipos não precisamos nos preocupar em que momento teremos que fazer isso.

6.2.2 Uso de ponteiros para ponteiros como parâmetros

Um caso especial de passagem de parâmetros por referência ocorre quando se quer passar um ponteiro para uma função que pode modificar seu conteúdo. A modificação do conteúdo de um ponteiro representa, na prática, mudar a posição da memória para qual ela aponta. Uma forma de obter isso é fazer com que a função passe a retornar um endereço. Assim, o parâmetro do comando `return` é o novo endereço para o qual o ponteiro apontará. Isso funciona caso apenas uma variável ponteiro seja modificada. Em situações em que mais do que um endereço tenha que ser atualizado não é possível fazer a atribuição como valor de retorno.

Para essas situações o que se faz é passar como parâmetro da função o endereço do próprio ponteiro e não o endereço por ele apontado. Isso é feito com o uso de uma estrutura chamada ponteiro para ponteiro, ou ponteiro duplo. Em um ponteiro duplo o valor recebido na função é o endereço da variável ponteiro e, com isso, alterações de seu conteúdo implicam na mudança do endereço apontado por aquele ponteiro. A sintaxe para esse tipo de passagem leva em consideração o fato de estarmos tratando de endereços. Assim, na chamada da função temos:

```
// chamando a função passando o endereço do ponteiro pont
var_x = func(&pont, ...);  // supondo que pont aponta para "int"
....
```

Assim, para a função `func` foi passado o endereço de `pont` e não seu conteúdo, que seria o de alguma variável inteira. Dentro da função temos a variável `dp`, que recebe o endereço do ponteiro `pont`. Como `dp` é um ponteiro para um ponteiro, na lista de parâmetros isso é representado como `int **dp`. O uso dessa dupla indireção, dentro da função, é feito na forma apresentada no código seguinte, em que apresentamos também erros que podem ser cometidos ao não se observar quais variáveis estamos de fato tratando.

```
tipo_f func (int **dp; ...) // dp recebe o endereço de pont
{int x, y, *ender;

  x = *dp;   // ERRADO, pois atribui um endereço para x
  x = **dp;  // atribui a x o valor do inteiro apontado por pont
  y = 10;
  ender = &y; // faz ender apontar para y
  *dp = *ender; // ERRADO, pois atribui um inteiro para ponteiro
  *dp = ender; // atribui o endereço de y para pont
}
```

O comando na linha final, em particular, é o uso pretendido para ponteiros duplos. O que ele faz, na prática, é mudar o conteúdo de `pont`, de modo a que ele passe a apontar para o endereço da variável `y`. Embora isso seja importante quando estivermos trabalhando com variáveis dinâmicas (capítulo 8), uma atribuição como a feita nesse exemplo é perigosa, uma vez que a variável `y` é local à função e pode ter seu espaço liberado após a saída da função.

6.3 Tipos de subprogramas

Nossa discussão até aqui indicou que temos subprogramas que retornam um valor explicitamente a quem os ativou e subprogramas que não fazem isso. Boa parte das linguagens de programação apresenta estruturas sintáticas distintas para esses dois tipos de subprogramas. Por exemplo Pascal apresenta *function* e *procedure*, enquanto Fortran apresenta *functcion* e *subroutine*, respectivamente para subprogramas com retorno e sem retorno explícito.

Como em C todo subprograma é tratado como uma função, não faz sentido ter denominações distintas. Apesar disso é interessante diferenciar funções que retornam um valor explícito daquelas que não o fazem, mostrando como modelo e algoritmo do programa são afetados. A Tabela 6.1 sumariza essas questões, principalmente o formato possível de chamada e a existência de variável de retorno. Essas características são gerais e independem da linguagem usada, mesmo que usem nomes distintos para esses dois tipos de subprogramas.

Para entender melhor as diferenças entre essas formas de subprogramas, bem como aspectos de escopo e passagem de parâmetros, vamos trabalhar agora sobre um exemplo mais complexo, que é o de resolução de um sistema de equações pela regra de Cramer (limitado a três variáveis e equações), calculando os determinantes pelo método de Sarrus.

Tabela 6.1: Diferenças entre subprogramas com e sem retorno de resultado

Aspecto	Com retorno explícito	Sem retorno explícito
Forma de chamada	Aparece sempre do lado direito de uma atribuição	Aparece sempre como um comando isolado
Tipos de parâmetros	Tipicamente apenas com passagem por valor	Pelo menos um deve ser com passagem por referência
Quantidade de respostas	Apenas um valor é retornado como resultado	Todos os parâmetros passados como referência são respostas
Variável de retorno	Variável à esquerda da atribuição armazena a resposta	Não usa variáveis de retorno
Variáveis globais	Sofrem modificação caso recebam alguma atribuição	Sofrem modificação caso recebam alguma atribuição

---

Exemplo

O problema

Como enunciado, nosso problema aqui é escrever um programa que resolva sistemas lineares de duas ou três equações, usando a regra de Cramer.

O modelo

Para resolver esses sistemas usaremos a regra de Cramer transformando o sistema em determinantes, ou seja, dado:

$Ax = b$ em que A é uma matriz com os coeficientes do sistema e b o vetor com as soluções das equações

temos que o determinante de A (Δ) e os determinantes de A_{x_i} (Δ_{x_i}), devem ser calculados para determinar os valores de x_i ($x_i = \frac{\Delta_{x_i}}{\Delta}$), resolvendo o sistema com as seguintes condições:

- O sistema tem solução única se $\Delta \neq 0$;
- O sistema não tem solução se $\Delta = 0$ e $\Delta_{x_i} \neq 0$;
- O sistema tem infinitas soluções se $\Delta = 0$ e $\Delta_{x_i} = 0$, para todo x_i;

O cálculo de cada determinante pode ser feito por meio de uma regra simples, método de Sarrus, em que se multiplica as diagonais da matriz ampliada. Sua aplicação para a matriz ampliada dada por:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,1} & a_{3,2} \end{bmatrix}$$

Resulta em:

$$\Delta = \begin{array}{l} (a_{1,1} * a_{2,2} * a_{3,3}) + (a_{1,2} * a_{2,3} * a_{3,1}) + (a_{1,3} * a_{2,1} * a_{3,2}) \\ -(a_{1,1} * a_{2,3} * a_{3,2}) - (a_{1,2} * a_{2,1} * a_{3,3}) - (a_{1,3} * a_{2,2} * a_{3,1}) \end{array}$$

Além disso, para calcular Δ_{x_i} devemos substituir os coeficientes da coluna i na matriz A pelos valores correspondentes do vetor b.

Os algoritmos

Do modelo definido pode-se observar que serão necessários alguns subprogramas distintos, como leitura do sistema de equações, cálculo de Δ e cálculo dos Δ_{x_i}. Outros pontos, como o cálculo de x_i e impressão dos resultados, podem tanto ser colocados em subprogramas quanto no próprio programa principal. Nossa opção aqui será por deixar o programa principal o mais limpo possível, contendo preferencialmente apenas chamadas para os demais subprogramas (o que é uma técnica muito eficiente, devendo ser usada sempre que possível).

Assim, podemos definir um algoritmo em altíssimo nível dado por:

1. Leia o número de equações (2 ou 3);
2. Leia os coeficientes do sistema de equações;
3. Calcule o determinante (Δ);
4. Se $\Delta = 0$ indique que o sistema não tem solução única e termine a execução;
5. Como $\Delta \neq 0$, então para cada x_i faça:
   (a) Calcule o determinante Δ_{x_i};
   (b) Calcule $x_i = \frac{\Delta_{x_i}}{\Delta}$
6. Apresente os resultados.

Observando esse algoritmo vemos que necessitam de maior detalhamento as etapas de cálculo do determinante. Esse detalhamento deve ser apresentado a partir da equação apresentada para sistemas com três equações e sua redução para sistemas com duas equações. Como o cálculo do determinante deve ocorrer

várias vezes durante o processo de resolução, então podemos deixar esse módulo como um subprograma, ao qual teríamos que passar o número de equações e os coeficientes a serem usados (incluindo as matrizes alteradas com o vetor *b*). O algoritmo para o subprograma *Determinante* ficaria então:

1. Receba o tamanho da matriz e a própria matriz (*Mat*)
2. Se tamanho = 2

$$Determinante = (Mat_{1,1} * Mat_{2,2}) - (Mat_{1,2} * Mat_{2,1})$$

3. Senão

$$\begin{array}{rl} & (Mat_{1,1} * Mat_{2,2} * Mat_{3,3}) + (Mat_{1,2} * Mat_{2,3} * Mat_{3,1}) \\ Determinante = & +(Mat_{1,3} * Mat_{2,1} * Mat_{3,2}) - (Mat_{1,1} * Mat_{2,3} * Mat_{3,2}) \\ & -(Mat_{1,2} * Mat_{2,1} * Mat_{3,3}) - (Mat_{1,3} * Mat_{2,2} * Mat_{3,1}) \end{array}$$

4. Retorne o valor de Determinante

Da mesma forma faltam também as partes do algoritmo que fazem a leitura das equações e a inserção do vetor *b* nas colunas da matriz, para o cálculo dos Δ_{x_i}. Não detalharemos aqui esses subprogramas, apresentando-os diretamente no código, uma vez que são simples.

A única observação a ser feita é sobre a troca de uma coluna da matriz pelo vetor *b*. Se fizermos isso diretamente na matriz original estaremos destruindo o original e, portanto, perdendo os seus valores corretos. Desse modo precisamos inserir o vetor *b* numa cópia da matriz e passar essa cópia para a matriz que calcula o determinante. Uma estratégia para obter esse resultado é fazer uma chamada para um subprograma *TrocaCol*, passando-lhe uma cópia da matriz e, a partir desse programa fazer a chamada do subprograma *Determinante*. O valor retornado por esse último seria também repassado para o programa principal.

O programa

Uma primeira versão de um programa para resolver nosso problema aparece na figura 6.9. Esse programa, embora aparentemente correto, não funciona da forma esperada. Por exemplo, um sistema com as equações $x_1 + x_2 = 0$ e $x_1 - x_2 = 2$ apresenta como resposta os valores 1 e 0, em vez dos valores corretos 1 e -1.

Uma primeira hipótese seria um erro na chamada de *LeSistema*, em que os parâmetros relativos ao sistema de equações (matriz e vetor solução) são passados explicitamente por referência, esperando que ao concluir sua execução tenhamos na função *main* os valores corretos nas variáveis *Matriz* e *vetorB*. Essa passagem, entretanto, está correta e não é a causa de nosso erro.

```
#include "stdio.h"
float Determinante(float[3][3], int);// protótipos de Determinante
float TrocaCol(float[3][3], float[3], int, int); // e TrocaCol

void LeSistema (float *Mat, float *Vet, int tam)
{int i,j;
 float *Maux=Mat;

  for (i=0;i<tam;i++)
  { Mat = (Maux+i*3);
    for(j=0;j<tam;j++)
    { printf("Qual o valor do coeficiente %d,%d:\n",i,j);
      scanf("%f",&(*Mat)); Mat++;
    }
    puts("Qual o valor do vetor b para esta equação");
    scanf("%f", &(*Vet)); Vet++;
  }
}
```

Figura 6.9: Programa para solução de sistemas com 2 ou 3 equações

O erro, na verdade, surgiu por termos desconsiderado a forma em que C trata a passagem de vetores e matrizes para funções. No código apresentado se considerou que a chamada da função *TrocaCol* ocorre, de fato, com passagem de parâmetros por valor.

Infelizmente (ou felizmente, dependendo do ponto de vista) os compiladores C tratam estruturas do tipo vetor (de qualquer dimensão) como sendo um ponteiro para a primeira posição do vetor. Assim, vetores são sempre passados por referência. Isso faz com que a função *TrocaCol* troque, efetivamente, as colunas da matriz original, destruindo seu conteúdo inicial após o laço em que troca os elementos da matriz pelos elementos do vetor *b*.

Para corrigir esse problema podemos passar explicitamente uma cópia da matriz a partir da função *main*, ou fazer uma cópia da matriz dentro da função *TrocaCol*, como apresentado na figura 6.10. Nela o que se faz é copiar o conteúdo original da matriz contendo os coeficientes do sistema de equações para uma matriz auxiliar, que tem então uma coluna modificada antes de ser passada para a função *Determinante*, mantendo assim a matriz original.

Assim, devemos tomar bastante cuidado com o uso de variáveis na forma de estruturas homogêneas (vetores, matrizes, cadeias de caracteres) quando passadas como argumentos para funções. Devemos sempre nos lembrar que tais variáveis

```
float TrocaCol(float Mat[3][3], float b[3], int col, int tam)
{int i,j;

   for (i=0; i<tam; i++)
      Mat[i][col] = b[i];  // Faz a troca da coluna col na matriz
   return(Determinante(Mat, tam));// Calcula o determinante para
      // a matriz com a troca da coluna col e retorna esse valor
}

float Determinante(float M[3][3], int tam)
{int i, j;

 if (tam == 2)
    return (M[0][0]*M[1][1] - M[0][1]*M[1][0]);
  else
    return (M[0][0]*M[1][1]*M[2][2] + M[0][1]*M[1][2]*M[2][0] +
            M[0][2]*M[1][0]*M[2][1] - M[0][0]*M[1][2]*M[2][1] -
            M[0][1]*M[1][0]*M[2][2] - M[0][2]*M[1][1]*M[2][0]  );
}

main()
{int tamanho, i, j;
 float Matriz[3][3], vetorB[3], Delta, DeltaX[3];

  puts("Tamanho do sistema?");  scanf("%d",&tamanho); // Passo 1
  LeSistema(&(Matriz[0][0]), &(vetorB[0]), tamanho);  // Passo 2
  Delta = Determinante(Matriz, tamanho); // Passo 3
  if (Delta == 0)
     { puts ("Delta=0, Sistema não tem solução"); exit(0); }
  for (i=0; i < tamanho; i++)
  { DeltaX[i] = TrocaCol (Matriz, vetorB, i, tamanho); // Passo 4
    printf("X[%d] = %f\n",i,DeltaX[i]/Delta); // Passos 5 e 6
  }
}
```

Figura 6.9: Programa para solução de sistemas com 2 ou 3 equações - Parte 2

serão passadas na forma de um ponteiro para a sua primeira posição. Isso implica que na função chamada será perfeitamente possível alterar o conteúdo original da variável. Se quisermos preservar o conteúdo original, apesar de qualquer alteração

```
float TrocaCol(float Mat[3][3], float b[3], int col, int tam)
{int i,j;
 float auxmat[3][3];

   for (i=0; i<tam; i++)
   { for (j=0; j<tam; j++)   // Copia uma linha da matriz
        auxmat[i][j] = Mat[i][j];
     auxmat[i][col] = b[i];  // Faz a troca da coluna col
   }
   return(Determinante(auxmat, tam)); // Calcula o determinante
      // para a matriz modificada e retorna esse valor
}
```

Figura 6.10: Programa correto para solução de sistemas com 2 ou 3 equações

realizada na função, então teremos que passar uma cópia da estrutura, como feito no código apresentado na figura 6.9.

Para concluir essa discussão sobre passagem de parâmetros para funções devemos lembrar que o uso implícito do endereço de uma variável ocorre apenas para estruturas compostas. No caso de variáveis simples, como inteiros, reais, caracteres ou mesmo uma posição específica de uma estrutura composta, o argumento passado é o seu conteúdo, caracterizando então a passagem por valor. Nesses casos, se quisermos usar o endereço da variável, teremos que passar seu endereço de forma explícita, como ponteiro.

6.4 Subprogramas recursivos

Apesar da grande variedade de aplicações que fazem uso de subprogramas na forma em que foram apresentados, existe uma outra forma de se trabalhar com subprogramas. Até agora trabalhamos com subprogramas iterativos, em que a resolução do problema é feita de forma direta, usando no máximo laços internos ao subprograma e chamadas para outros subprogramas. A outra forma possível de se trabalhar é fazendo uso de subprogramas recursivos.

Subprogramas recursivos trabalham repetindo as atividades previstas para o subprograma, porém com chamadas para o próprio subprograma usando parâmetros novos. Esse tipo de chamada usa a propriedade de recursividade, que é a capacidade de definir a resposta de algum problema a partir de definições que envolvam indução finita e a solução conhecida para um problema equivalente. Isso ocorre, por exemplo, na definição recursiva de funções como a do fatorial, vista a seguir:

$$x! = \begin{cases} 1 & , \quad se \quad x = 1 \\ x.(x-1)! & , \quad se \quad x > 1 \end{cases}$$

A grande vantagem em definições recursivas é que podemos obter soluções simples para problemas teoricamente ilimitados e complexos. Essas soluções são construídas a partir de uma solução que seja conhecida para um caso limitado e uma regra de recorrência, que defina como o problema pode ser particionado para chegar ao caso limitado. A combinação da regra de recorrência e da solução conhecida permite, então, construir um procedimento simplificado para obter a solução para problemas de qualquer tamanho. Isso é o que ocorre, por exemplo, na especificação recursiva do cálculo de fatorial apresentada acima.

Em programação o conceito de recursividade é extremamente útil. Com algoritmos recursivos podemos diminuir muito o tamanho dos programas que escrevemos. Para tanto, temos apenas que ter uma definição recursiva para o problema que estamos resolvendo. Essa definição deve ter, como já visto, uma parte para a regra de recorrência e outra para a solução conhecida. Do mesmo modo, um programa construído a partir dessa solução terá que tratar separadamente cada uma dessas partes. Mais explicitamente, um programa recursivo deve ter:

1. Uma parte básica, que define a solução para o caso mais simples do problema a ser resolvido. Para o fatorial, isso seria o caso em que $x = 1$ (considerando que $x = 0$ é inerentemente trivial);

2. Uma parte recursiva, que define a regra de recorrência, ou como a solução para um problema mais geral é definida a partir da parte básica.

Para o exemplo do fatorial, considerando-se que seu cálculo é possível apenas para números naturais, temos que a solução trivial ocorre para os valores 0 ou 1. Em ambos os casos é sabido que o valor do fatorial desses números é igual a 1. Para a parte recursiva temos como regra de recorrência a parte inferior da definição recursiva do fatorial, ou seja, $x! = x.(x-1)!$. Isso implica que uma função para o cálculo do fatorial teria que incluir um teste para saber o valor do fatorial a ser calculado e apresentar as duas soluções possíveis, em que uma faria uso de recursão.

Na figura 6.11 temos o código de uma função recursiva para o cálculo do fatorial.

O funcionamento dessa função é o seguinte:

```
int fatorialRecursivo(int num)
{int fatorial;

   if ((num == 1) || (num == 0))
       fatorial = 1;
     else
       fatorial = num * fatorialRecursivo(num - 1);
   return (fatorial);
}
```

Figura 6.11: Função recursiva para o cálculo do fatorial.

1. A função é ativada de algum ponto do programa, com o parâmetro *num* passado por valor e, nessa descrição, suponhamos que ele seja igual a 3 inicialmente;

2. Testa-se o valor de *num* contra 1 ou 0. No caso de ser igual a um desses valores atribuiria-se o valor 1 para a variável *fatorial*, retornando-se esse resultado para o ponto do programa que fez a chamada;

3. Como *num* é igual a 3, o resultado a ser retornado é igual a 3 vezes o resultado da chamada da função *fatorialRecursivo*, agora tendo como parâmetro o valor 2 (vindo de $num - 1$);

4. A função é ativada agora com o parâmetro *num* igual a 2, repetindo-se o teste e resultando em que o valor a ser retornado é igual a 2 vezes o resultado da função *fatorialRecursivo* com parâmetro igual a 1;

5. Nessa ativação da função o valor de *num* é igual a 1 e com a repetição do teste é retornado o valor 1 para o ponto do programa que a chamou, que é exatamente a instância anterior da função *fatorialRecursivo* (em que *num* era igual a 2);

6. Encerra-se então o cálculo do passo 4, multiplicando o resultado obtido em 5 (fatorial=1) por 2, retornando-se esse produto como o resultado dessa ativação de *fatorialRecursivo* para a instância anterior dessa função (em que *num* era igual a 3);

7. O resultado retornado pelo passo 6 (fatorial=2) é então multiplicado por 3, para finalizar a execução do passo 3, e retorna-se então o resultado desse produto (fatorial=6) como sendo o fatorial para a função que iniciou a primeira ativação de *fatorialRecursivo* (passo 1).

Deve-se observar que o código apresentado para a função fatorial não é tão imediato quanto o código visto inicialmente no capítulo 1. Uma diferença importante é a de que aqui o valor do fatorial é determinado com uma sequência de chamadas e posteriores retornos da função *fatorialRecursivo*, enquanto que na versão original tínhamos um laço de repetição cujo corpo era executado um certo número de vezes, até termos o valor do fatorial. Em última análise, a essência de programas recursivos envolve a chamada de uma função por ela mesma, em a ação repetitiva sobre um objeto é feita não por meio de comandos de repetição, mas sim pela definição recursiva da ação.

Desse modo resulta uma primeira impressão, que veremos ser errada, de que o uso de recursão torna mais complicadas as soluções computacionais de problemas. Essa primeira impressão vem, na realidade, pelo fato do cálculo do fatorial possuir uma solução iterativa muito (mas muito) simples. Assim, a solução recursiva, embora elegante, parece exagerada pelo tamanho do problema. Seria como atirar numa mosca com munição de caçar elefantes.

Existem, entretanto, inúmeros problemas para os quais a solução recursiva é, não apenas a mais elegante, mas também a mais rápida e eficiente, principalmente na área de organização e recuperação de informações. Outra categoria de aplicações é a que envolve a solução de problemas de sequência de operações com ordenação lógica (provas de teoremas e alguns jogos). Um desses problemas é o da "Torre de Hanói", que é um quebra-cabeças em que vários discos de diâmetros diferentes estão colocados em um dos três postes do jogo, como na figura 6.12. O problema inicia-se colocando-se todos os discos no poste da esquerda, em ordem de tamanho, com o maior disco na base e decrescendo até o topo da pilha de discos. O objetivo é colocar todos os discos no poste da direita, por meio de operações que respeitem às seguintes regras:

1. Apenas podemos movimentar um único disco de cada vez;
2. Podemos movimentar apenas o disco que está no topo da pilha do poste;
3. Nunca se pode colocar um disco de diâmetro maior sobre um de menor diâmetro.

6.4.1 Problema da Torre de Hanói

A solução mecânica (humana) para a Torre de Hanói é simples. Para uma torre com n discos renomeamos os postes como sendo os postes **origem**, **destino**, e **auxiliar**, respectivamente como aqueles que contêm a pilha original de discos, que conterão a pilha final e que serão usados temporariamente na movimentação.

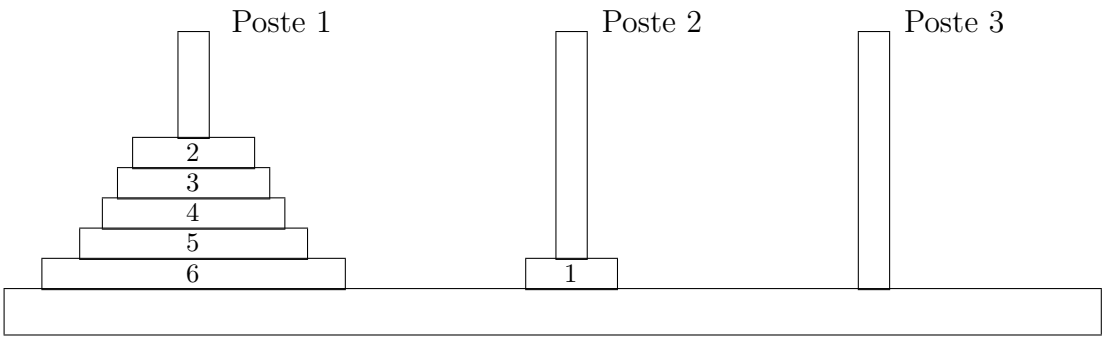

Figura 6.12: Torre de Hanói com disco 1 já movimentado.

Como o problema de movimentação é, na realidade, um problema de contagem segundo o binômio de Newton, sabe-se de antemão que o número mínimo de movimentos é igual a $2^n - 1$ para n discos. Sabemos ainda que para um número par de discos, o primeiro movimento deve levar o disco 1 do poste **origem** para o poste **auxiliar**. Da mesma forma, se temos um número ímpar de discos devemos levar inicialmente o disco 1 para o poste **destino**. Os movimentos seguintes também seguem uma ordem estrita de posições, procurando fazer com que a sequência deles leve até a solução do problema.

No diagrama da figura 6.12 estamos considerando o poste 1 como origem, o poste 2 como auxiliar e o poste 3 como destino. Os próximos cinco passos, considerando a posição disposta naquela figura, seriam levar o disco 2 do poste origem para o destino, seguido de levar o disco 1 do auxiliar para o destino e o disco 3 do origem para o auxiliar, quando então levamos o disco 1 do destino para o origem e, finalmente, o disco 2 do destino para o auxiliar, chegando na configuração apresentada na figura 6.13.

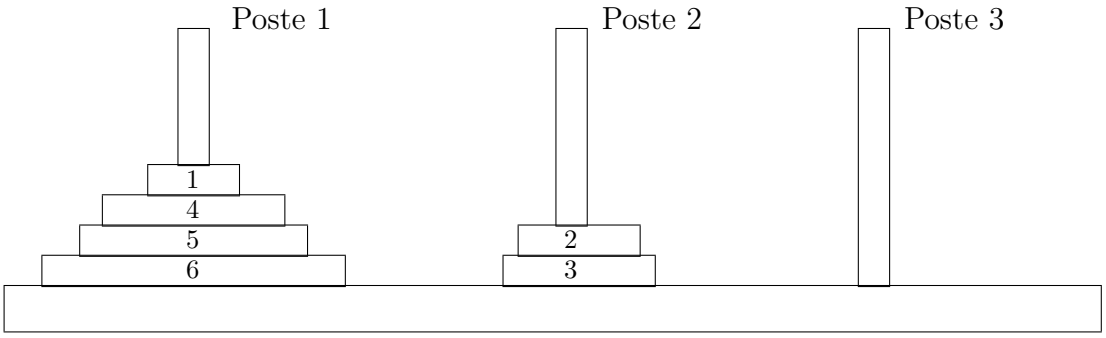

Figura 6.13: Torre de Hanói após seis movimentos.

O problema que aparece nesse momento é como escrever um programa que apresente, por exemplo, a sequência de passos para movimentar n discos do **Poste 1** para o **Poste 3**. Como seria uma solução iterativa para o problema?

A saída a ser apresentada pelo programa é trivial, consistindo em mensagens do tipo "Levei o disco k do poste i para o poste j". O primeiro movimento também é trivial, bastando saber se o total de discos é par ou ímpar. As coisas complicam a partir do segundo movimento, quando temos que saber qual disco iremos movimentar e entre quais postes a partir dos movimentos realizados anteriormente. Um algoritmo para isso considera a paridade do disco movido no passo anterior, resultando nas seguintes opções para o próximo movimento:

1. Se o disco movido era par, então mova um disco do poste não utilizado nesse movimento e o coloque sobre esse disco;
2. Se o disco movido era ímpar, então o movimento não envolverá o poste em que foi colocado, levando um disco de um dos outros postes para o outro, de forma a não colocar um disco maior sobre outro menor.

Esse procedimento prossegue até que todos os discos estejam sobre o poste destino. A solução iterativa necessita que mantenhamos um registro de quais postes foram usados, quais discos foram movimentados e quantos e quais discos estão em cada poste. Isso aparentemente não é tão complexo e pode ser feito em poucas linhas de código, como o mostrado na figura 6.14.

Observe-se que essa solução está bastante otimizada e confusa quanto aos índices dos vetores que controlam os postes (*to, fr, sp*). Outra otimização nesse programa envolve a macro *ALLO*, que faz a alocação de posições para os postes, na quantidade de discos a serem usados. Como estruturas dinâmicas estão fora do contexto desse material, acreditemos nesse momento que as chamadas para *ALLO* apenas "definem" os vetores com os tamanhos corretos para o número de discos usados.

Embora existam soluções iterativas eficientes, percebe-se que o raciocínio por trás delas é um tanto quanto confuso. Uma solução recursiva para o mesmo problema é fundamentalmente elegante e não necessita de nenhuma das informações indicadas no parágrafo anterior ou no código da figura 6.14.

A solução recursiva parte da identificação de que o problema com um disco tem solução trivial, consistindo apenas na movimentação deste disco do poste origem para o poste destino. Observando-se então que a solução para uma quantidade n qualquer de discos pode ser obtida considerando primeiro que temos que mover $n-1$ discos do poste origem para o poste auxiliar, chegando então ao problema trivial de um disco, e depois movimentando os mesmos $n-1$ discos do poste

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

#define ALLO(x) { x = (int *)malloc((n+3) * sizeof(int)); }

main(int argc, char *argv[])
{ int i, *a, *b, *c, *p, *fr, *to, *sp, n, n1, n2;

 n = atoi(argv[1]); // número de discos vem da linha de comando
 n1 = n+1;
 n2 = n+2;
 ALLO(a);     ALLO(b);     ALLO(c); // cria os vetores a, b e c
 a[0] = 1;
 b[0] = c[0] = n1;
 a[n1] = b[n1] = c[n1] = n1;
 a[n2] = 1;   b[n2] = 2;   c[n2] = 3;
 for(i=1; i<n1; i++) {   // coloca os discos no vetor a
   a[i] = i; b[i] = c[i] = 0;
 }
 fr = a;   // diz que o vetor a é o poste de origem
 if(n&1) { to = c; sp = b; } // verifica se n é par (AND bitwise)
   else  { to = b; sp = c; }
 while(c[0]>1) {
   printf("disc %d from %d to %d\n",i=fr[fr[0]++],fr[n2],to[n2]);
   p=sp;
   if ((to[--to[0]] = i)&1) // testa se disco a ser movido é par
       { sp=to;
         if (fr[fr[0]] > p[p[0]]) // testa o movimento a fazer
             {to=fr; fr=p;}
           else
             to=p;
       }
     else
       { sp=fr; fr=p; }
 }
}
```

Figura 6.14: Torre de Hanói iterativa

auxiliar para o destino, podemos chegar à seguinte solução recursiva:

$$
\textit{torre} = \begin{cases} \textit{se 1 disco} & \textit{1. mover do poste atual para seu destino} \\ \textit{se n discos} & \begin{cases} \textit{1. resolver torre com n-1 discos do poste atual para o poste auxiliar} \\ \textit{2. resolver torre com um disco} \\ \textit{3. resolver torre com n-1 discos do poste auxiliar para o poste destino} \end{cases} \end{cases}
$$

Esse pseudoalgoritmo leva, com alguma facilidade, ao programa recursivo para a Torre de Hanói visto na figura 6.15.

```
#include <stdio.h>
hanoi(int n, int o, int a, int d)
{   if (n == 1)  /* se apenas um disco a solução é trivial  */
 /*Passo 0*/  printf ("move disco %d de %d para %d\n",n,o,d);
        else  /* senão adota a solução recursiva  */
 /*Passo 1*/  { hanoi(n-1, o,d,a);
 /*Passo 2*/    printf ("move disco %d de %d para %d\n",n,o,d);
 /*Passo 3*/    hanoi(n-1, a,o,d);
              }
 /*Passo 4*/
}

main(int argc, char *argv[])
{int n;  // n é o número inicial de discos na torre
  n = atoi(argv[1]); // convertido do valor na linha de comando
  hanoi(n,1,2,3); // muda n discos do poste 1 para o poste 3
}
```

Figura 6.15: Torre de Hanói recursiva

Observando a solução recursiva fica evidente que ela é mais curta e elegante. Uma única nota a ser feita é a de que o segundo passo da solução para n discos (com $n > 1$) foi alterado para já conter imediatamente a movimentação do disco e não uma nova chamada da função *hanoi*. Isso serve como uma economia no número de chamadas da função, embora pudesse ser feita com uma nova chamada sem nenhum problema.

Embora a função *hanoi* seja bastante simples, é interessante fazermos um

pequeno rastreamento de seu funcionamento para, por exemplo, uma torre com 4 discos. A primeira metade dos movimentos é mostrada na figura 6.16, incluindo o conteúdo das variáveis da função *hanoi* a cada chamada e as mensagens impressas.

Posição no programa	Valor de variáveis	Saída
main, chama hanoi	n=4	
hanoi(1), passo 1	n=4, o=1, a=2, d=3	
hanoi(2), passo 1	n=3, o=1, a=3, d=2	
hanoi(3), passo 1	n=2, o=1, a=2, d=3	
hanoi(4), passo 0	n=1, o=1, a=3, d=2	move disco 1 de 1 para 2
hanoi(4), passo 4	n=1, o=1, a=3, d=2	
hanoi(3), passo 2	n=2, o=1, a=2, d=3	move disco 2 de 1 para 3
hanoi(3), passo 3	n=2, o=1, a=2, d=3	
hanoi(4a), passo 0	n=1, o=2, a=1, d=3	move disco 1 de 2 para 3
hanoi(4a), passo 4	n=1, o=2, a=1, d=3	
hanoi(3), passo 4	n=2, o=1, a=2, d=3	
hanoi(2), passo 2	n=3, o=1, a=3, d=2	move disco 3 de 1 para 2
hanoi(2), passo 3	n=3, o=1, a=3, d=2	
hanoi(3a), passo 1	n=2, o=3, a=1, d=2	
hanoi(4b), passo 0	n=1, o=3, a=2, d=1	move disco 1 de 3 para 1
hanoi(4b), passo 4	n=1, o=3, a=2, d=1	
hanoi(3a), passo 2	n=2, o=3, a=1, d=2	move disco 2 de 3 para 2
hanoi(3a), passo 3	n=2, o=3, a=1, d=2	
hanoi(4c), passo 0	n=1, o=1, a=3, d=2	move disco 1 de 1 para 2
hanoi(4c), passo 4	n=1, o=1, a=3, d=2	
hanoi(3a), passo 4	n=2, o=3, a=1, d=2	
hanoi(2), passo 4	n=3, o=1, a=3, d=2	
hanoi(1), passo 2	n=4, o=1, a=2, d=3	move disco 4 de 1 para 3
hanoi(1), passo 3	n=4, o=1, a=2, d=3	
.....		

Figura 6.16: Execução da Torre de Hanói para 4 discos. Os valores entre parenteses indicam qual instância de chamada da função está sendo executada naquela linha.

6.4.2 Problemas recursivos clássicos

Outro exemplo clássico de jogos com solução recursiva é o da movimentação do cavalo num tabuleiro de xadrez, percorrendo todas as suas casas sem repetição.

Aqui a solução também passa por reduzir o problema até uma solução conhecida, que é o de percorrer uma posição do tabuleiro. Uma das formas de se fazer o caminho é partir de uma dada posição do tabuleiro e, a partir dela, determinar as posições possíveis para um movimento do cavalo, que seriam armazenadas num vetor de soluções. Para uma dessas posições (um método eficiente de escolha é pegar o movimento que leva para a posição mais a esquerda da atual) chama-se a função *moveCavalo(posicao)*, que marca a posição atual na matriz como já percorrida e volta a determinar os movimentos possíveis (que vão para posições ainda não percorridas), chamando recursivamente a função *moveCavalo*, até que se complete todos os movimentos ou se chegue num caminho impossível.

Se todas as posições foram preenchidas termina-se a execução. Caso se chegue a um caminho impossível realiza-se o chamado movimento de *backtracking*, em que se volta um passo no processo de solução (volta o cavalo para a posição anterior, nesse problema)e tenta-se um movimento diferente daquele que havia sido tentado. O processo de *backtracking* é terminado se todos os possíveis caminhos forem testados e verificados como impossíveis.

Em mais um exemplo ligado ao xadrez temos o chamado problema das N rainhas, em que se deve colocar N rainhas num tabuleiro NxN de tal forma que nenhuma das rainhas esteja em posição de ser atacada por (ou atacar) outra rainha. Aqui a técnica recursiva trabalha procurando colocar uma rainha por vez, até conseguir colocar todas as N rainhas. O processo recursivo usa também a técnica de *backtracking* (caminho de volta) para retroceder até uma posição segura toda vez que se chega numa situação de conflito (ataque) entre duas rainhas.

A figura 6.17 apresenta parte da solução recursiva para o problema das rainhas. Nela ficam faltando duas funções, uma para apresentar a solução encontrada (*drawboard*) e outra para verificar se existem rainhas em situação de ataque (*good*). Essas duas funções são fáceis de implementar.

Além de problemas da área de jogos, que acabam sendo tipicamente recursivos com o uso de *backtracking*, o princípio da recursividade é aplicado em uma gama bastante ampla de problemas efetivamente mais "sérios" de computação.

Um desses casos é o da notação para expressões matemáticas. Normalmente usamos a chamada notação **infixa**, por colocar os operadores entre os operandos, como em $A + B$. Existem, entretanto, duas outras notações que ou colocam os operadores após o par de operandos (**pós-fixa**) ou antes deles (**prefixa**).

```
#include <stdio.h>

int board[40][40], boardSize;

try(int n) // função recursiva que coloca uma rainha no tabuleiro
{int i;
  for (i=0; i<boardSize; i++)
  { board[n][i] = 1; // coloca rainha em (n,i)
    if ((n==boardSize-1) && (good() == 1))
        return (1) // encontrou solução
      else
        if ((n<boardSize-1) && (good()== 1) && (try(n+1)== 1))
            return (1)
    board[n][i] = 0; // faz o backtracking aqui
  }
  return (0); // se não encontrou solução
}

main(int argc, char *argv[])
{int i, j;
   boardSize = atoi(argv[1]);  /* lê o tamanho do tabuleiro */
   for (i=0; i<boardSize; i++)
      for (j=0; j<boardSize; j++)
         board[i][j] = 0;
   if (try(0) == 1) // chama a função recursiva, mostrando a
       drawboard();    // resposta se encontrou solução
}
```

Figura 6.17: Problema das N rainhas.

Em particular, a notação pós-fixa também é conhecida como notação polônesa reversa, em contraposição à notação prefixa que é chamada polonesa em homenagem ao seu criador, o matemático polonês Jan Lukasiewicz. A vantagem no uso dessas notações é que elas não necessitam da presença de separadores de precedência, como os parenteses que usamos em expressões complexas. Um exemplo dessa situação é visto a seguir:

`A+B*(C-D)` $\equiv$ `ABCD-*+`

A conversão entre essas notações é importante, por exemplo, nos processos de compilação de um programa ou de tradução de textos escritos em linguagens naturais. O processo de conversão nesses caso é tipicamente resolvido usando chamadas recursivas, como mostrado na figura 6.18, em que o procedimento *lei-*

aToken faz a leitura de um elemento da expressão (o *token*) e, a partir do que for lido pode fazer uma chamada recursiva para ele próprio ou então realizar operações sobre a saída da conversão.

1. Leia *token*
2. Se *token* for um operando escreva-o e chame *leiaToken*
3. Senão faça:
   (a) Se operador tem precedência mais alta do que aquele que está na pilha ou a pilha está vazia ou for um '(', então guarde o operador na pilha
   (b) Senão faça:
      i. Se operador tem precedência menor que aquele que está na pilha, então enquanto isso for verdade, retire o operador que está no topo da pilha escrevendo-o na saída, guardando então o operador lido na pilha
      ii. Se operador for um ')', então retire todos os operadores da pilha, escrevendo-os na sequência em que forem retirados, até que se chegue ao próximo '(', descartando os parenteses
   (c) Chame *leiaToken*

Figura 6.18: Algoritmo da função *leiaToken* para conversão da notação infixa para pós-fixa.

Outro problema ligado a estruturas de dados é o de caminhos em árvores. Árvores são estruturas de dados relativamente simples, usadas para reduzir a complexidade de problemas de ordenação e busca de grandes volumes de dados. O nome da estrutura vem do fato de que nela os dados são armazenados numa forma cuja representação gráfica lembraria as ramificações de troncos e folhas de uma árvore real. Uma árvore binária (cada tronco se divide no máximo em outros dois) típica possui nós que contém um valor (o dado propriamente dito), um indicador para o elemento da árvore imediatamente a direita do nó e outro para o elemento a esquerda.

Assim, em um procedimento para achar o elemento mais a direita da árvore basta começarmos de um ponto conhecido (a raiz) e chamarmos esse procedimento recursivamente com um novo ponto de partida, que seria dado pelo elemento indicado como estando a direita do nó atual. A figura 6.19 mostra um trecho de código em C para caminhar até o nó mais a direita de uma árvore, usando o tipo ponteiro como método para a indicação dos nós a direita e a esquerda.

```
struct no_arvore { // estrutura de dados para a árvore
  tipo_no elem;    // campo com a informação propriamente dita
  struct no_arvore *no_dir; // ponteiros para outros nós
  struct no_arvore *no_esq; // da árvore
}

tipo_no encontra_no (struct no_arvore *no) // função recursiva
// que localiza o elemento mais a direita da árvore
{tipo_no resp;  // variável que armazena a resposta da função

  if (no->no_dir) // se ainda tem caminho a direita chama função
      resp = encontra_no (no->no_dir);
    else // senão temos a resposta
      resp = no->elem;
  return (resp); // retorna a resposta para quem chamou a função
}
```

Figura 6.19: Função para localizar elemento mais a direita em uma árvore

Com estes exemplos de recursão encerramos esse capítulo. Temos agora condições de escrever programas de qualquer complexidade, se bem que ainda com o uso de alguns modelos não tão eficientes para determinadas situações, como por exemplo para a manipulação de dados que envolvam informações de tipos diferentes, como nome, endereço, peso, altura e data de nascimento, mas que digam respeito ao mesmo elemento (uma certa pessoa, por exemplo).

Nos próximos capítulos passaremos a estudar tipos de dados mais complexos do que os que temos utilizado até então, com o uso de registros e apontadores. Ao usarmos esses tipos em conjunto com subprogramas e as estruturas de controle já apresentadas poderemos escrever qualquer tipo de programa, em qualquer tipo de linguagem do paradigma imperativo, além de termos condições de entender rapidamente os aspectos de programação nos demais paradigmas.

EXERCÍCIOS

1. Reescreva os programas apresentados nos exercícios dos capítulos 2 e 3, usando subprogramas para executar as tarefas requisitadas.

2. Escreva um programa que leia um número inteiro e o escreva por extenso (como uma máquina de preencher cheques faz, por exemplo).

3. Escreva um programa que leia N números, armazene-os em um vetor e depois os ordene, isso é, os coloque em ordem crescente.

4. Escreva um programa que possa ser usado como apoio ao ensino de física do ensino médio (apenas exercícios de cinemática), em que o aluno entraria com o tipo de problema e os dados de entrada. Seu programa deve apresentar a resposta para esse problema. Considere as variações possíveis de problemas e de quais seriam os dados de entrada.

5. Considere o problema do caminho do cavalo, apresentado na seção 4.4.2. Escreva um programa que, dado o tamanho do tabuleiro, apresente uma solução para o caminho (se houver) ou que diga que não existe solução para tabuleiro daquele tamanho.

6. Escreva um programa que, dada uma expressão em notação pós-fixa, apresente o resultado obtido com a avaliação da mesma.

Capítulo 7

Estruturas heterogêneas de dados

Durante o capítulo 2 apresentamos os tipos básicos de dados e também uma forma mais simples de organização coletiva de dados representada pelas estruturas homogêneas. Embora tais estruturas permitam resolver uma grande quantidade de problemas, existem situações em que precisamos misturar tipos de dados diferentes dentro de uma única estrutura, que seria associada a um único nome simbólico, o que não é possível com as estruturas homogêneas.

Um exemplo típico de situação em que isso ocorre é quando temos que armazenar e manipular dados relativos aos registros de pessoas dentro de uma organização. Tais dados envolvem diferentes tipos básicos, como cadeias de caracteres para armazenar nome da pessoa e partes do endereço, ou tipos numéricos para guardar idade, salário, etc. Esses dados de tipos diferentes não seriam normalmente associados, mas como todos dizem respeito a um mesmo elemento - uma pessoa da organização - então é interessante que tenhamos um mecanismo para associá-los numa única estrutura.

Essa estrutura é usualmente conhecida como estrutura heterogênea, uma vez que é usada para agrupar tipos de dados diferentes. A grande vantagem no uso de tais estruturas é permitir que os dados possam ser organizados de forma mais clara e próxima daquela que nós trabalhamos. No restante desse capítulo nos ocuparemos de mostrar porque necessitamos de estruturas heterogêneas e como podemos construí-las em uma linguagem estruturada (como C). Adicionalmente apresentaremos o conceito e forma de uso de arquivos. De modo simplificado, arquivos são formas alternativas de entrada ou saída de dados que podem ser compreendidos como sendo estruturas heterogêneas. Assim os examinaremos na parte final deste capítulo.

7.1 Aplicações de estruturas heterogêneas

Antes de examinar em que situações é desejável, ou imprescindível, usar estruturas heterogêneas precisamos entender o que elas de fato representam na memória. Quando declaramos uma variável como sendo do tipo de uma estrutura qualquer estamos, na prática, reservando na memória espaço suficiente para acomodar todos os tipos básicos nela definidos. Esse espaço também estaria organizado de forma bem definida, permitindo que o acesso a qualquer dos tipos básicos seja feito de maneira simples e direta.

Essa organização é estabelecida, durante a compilação, pela distinção entre a estrutura e seus componentes. O acesso a uma estrutura sempre ocorre por meio de seus componentes, sempre de forma individual a cada um deles. Esse acesso, assim como o endereço de uma casa (rua e número), ocorre pela nomeação simultânea da estrutura e do componente a ser acessado.

Com o uso de estruturas heterogêneas criamos todas as condições para escrever programas de qualquer complexidade, operando sobre qualquer tipo de dados e informações.

EXEMPLO

Considere uma empresa que mantém registros de seus empregados, usando-os para calcular sua folha de pagamento considerando número de horas trabalhadas, valor do salário base por hora, horário de trabalho, faixas de desconto de imposto de renda, número de dependentes, estado civil, tabela de descontos extraordinários (pensões, empréstimos, etc.). O programa que faz o processamento da folha de pagamento deve apresentar também o nome do empregado. Que tipos de dados devem ser organizados para gerar o registro de um empregado nessa empresa?

É natural que valores como salário base e faixas de desconto devem ser representados por números reais, assim como os descontos extraordinários. Além dessas variáveis devem ser acrescentadas outras para armazenar valores intermediários e até mesmo o salário líquido do empregado.

O número de horas trabalhadas deve ser um inteiro ou um real (assumindo o pagamento de frações de horas). Da mesma forma temos o número de dependentes e o número de registro (identificação do empregado) como valores inteiros.

O nome do empregado e seu estado civil são cadeias de caracteres. Outros campos com cadeias de caracteres podem envolver datas (nascimento, admissão, pagamento, etc.).

O conceito de estrutura heterogênea permitiria reunir todas essas informações em um único pacote, em que cada um desses dados seriam campos de um registro.

A figura 7.1 apresenta alguns dos campos dessa estrutura.

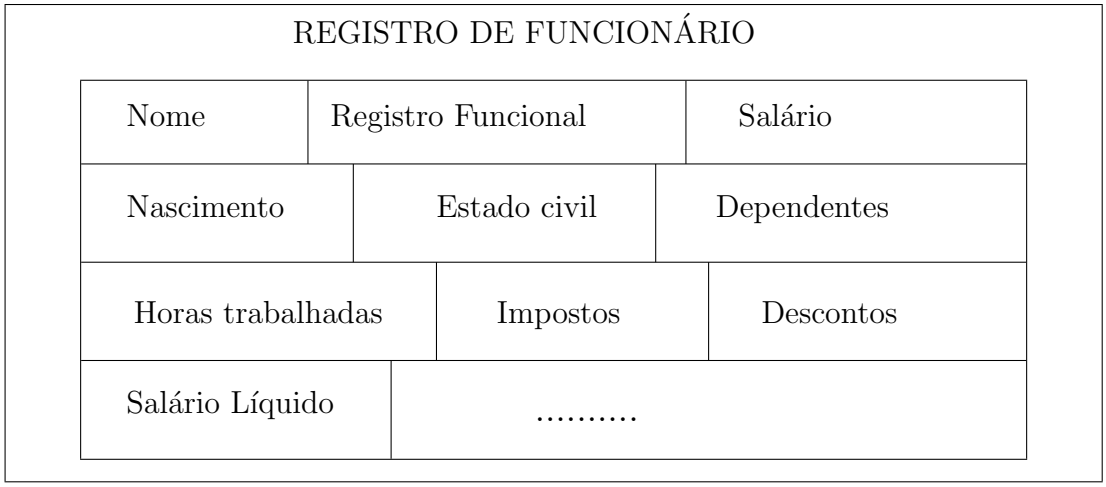

Figura 7.1: Estrutura para registro de um empregado

Deste exemplo podem ser feitas algumas observações, suficientemente gerais, sobre a organização de estruturas heterogêneas. Elas são:

1. O objetivo de reunir dados de tipos distintos em uma única estrutura é o de permitir concentrar nela todas as informações relativas a algum elemento do sistema;

2. O acesso aos dados individuais dentro de uma estrutura é possibilitado pela a alocação de espaços bem definidos na memória;

3. Para permitir uma organização correta dos dados na memória é preciso que cada um deles esteja claramente declarado e denominado, de forma a permitir o acesso individualizado aos mesmos;

4. Os componentes internos de uma estrutura são denominados **campos da estrutura**, sendo que os mesmos podem ser tipos básicos, compostos homogêneos ou mesmo outras estruturas heterogêneas previamente definidas.

Essas observações são essenciais para se entender quando e como devemos usar estruturas heterogêneas. O que resta no momento é apresentar um conjunto de situações em que a aplicação delas se torna a solução mais simples e eficiente. Isso é feito, nas próximas páginas, por meio de alguns exemplos de como estruturas são definidas na linguagem C.

7.2 Estruturas heterogêneas em C

Como apresentado no início desse capítulo, uma estrutura heterogênea é apenas uma forma simplificada e organizada para declarar e usar valores de tipos diferentes que tenham relação entre si. O uso dessas estruturas envolve, portanto, dois momentos distintos. Um, em que se faz a declaração da estrutura (e também de variáveis que serão do tipo dessa estrutura), e outro em que se define formas de manipulação dessas variáveis.

7.2.1 Declaração de uma estrutura

Em C a declaração de estruturas heterogêneas ocorre com a definição de uma ***struct***. Em uma *struct* o que temos é a definição de um nome para a estrutura, bem como a definição dos elementos que a compõem e quais são seus tipos. A figura 7.2 apresenta a declaração de uma estrutura em C para parte dos dados listados no exemplo apresentado na figura 7.1.

```
1   struct funcionario { // define a estrutura chamada "funcionario"
2     char nome[50];     // com os campos aqui listados
3     int reg;
4     char estcivil[8];
5     char nasc[15];
6     int dependentes;
7     float salBase;
8     float descontos;
9     int horastrab;
10   } emp, listaemp[2000]; // declara uma variável do tipo vetor de
11   // "funcionario" para armazenar os registros de todos empregados
12   // e outra variável do tipo "funcionario" p/ um empregado
```

Figura 7.2: Estrutura (simplificada) para registro de um empregado

A partir da declaração de uma estrutura é possível declarar variáveis daquele tipo, como o exemplo a seguir para a estrutura que acabamos de definir:

```
struct funcionario um_empregado, outro_empregado;
```

A linguagem C permite, também, identificar a estrutura como sendo um novo tipo de dados. Isso é feito com o uso do comando `typedef`. Esse comando necessita de dois parâmetros, que são a especificação do tipo a ser identificado e o nome pelo qual será conhecido. Assim, na Figura 7.3 pode ser visto o uso de `typedef` para o exemplo anterior. Nessa nova declaração se dá o nome `FUNC` para o tipo de dados definido pela estrutura *funcionario*.

```
1  typedef struct funcionario { // define um tipo para a estrutura
2    char nome[50];
3    // repete os campos listados anteriormente
4    int horastrab;
5  } FUNC;
6
7  FUNC emp, listaemp[2000]; // declara a variável "emp" e o vetor
8  // "listaemp" como sendo do tipo FUNC, que é um apelido para a
9  // estrutura criada
```

Figura 7.3: Exemplo do uso de *typedef* para denominar uma estrutura.

Deve ser observado aqui que o uso do *typedef* não altera a forma de manipular uma estrutura heterogênea. Seu uso permite apenas reduzir as declarações de variáveis daquele tipo, quando deixamos de usar *"struct nome_estrutura* sempre que fossemos declarar uma variável.

7.2.2 Manipulação de estruturas

O segundo momento no uso de uma estrutura heterogênea é a manipulação de seus conteúdos. A manipulação de estruturas ocorre somente por meio do acesso direto aos campos que a compõem. Esse acesso, como mencionado anteriormente, exige a identificação simultânea de qual estrutura se quer manipular e em qual campo dela ocorrerá essa manipulação. Essa dupla identificação ocorre, em C, pelo par

$$NomedaVariavel.NomedoCampo$$

como por exemplo em:

```
if (listaemp[n].salBase < 300.0) listaemp[n].descontos=0;
```

A manipulação dos conteúdos pode ser feita individualmente, como no exemplo acima, ou globalmente com o endereço da estrutura. Nesse último caso temos que fazer uso de ponteiros, usando uma sintaxe diferenciada. Nesse momento trabalharemos apenas com a forma convencional de acesso, que apresenta as seguintes características para as operações comuns em programas:

Leitura e escrita

As operações de entrada e saída devem ser realizadas de modo individual para cada elemento da estrutura, seguindo a formatação de cada tipo básico lido ou escrito.

```
scanf("%s %d",listaemp[j].nome, &listaemp[j].reg);
printf("%s %f",emp.nome, emp.salBase);
```

Atribuição de uma estrutura

A atribuição de uma estrutura para uma variável de seu tipo pode ser feita pela atribuição separada de cada um de seus campos ou pela atribuição direta entre variáveis do mesmo tipo de estrutura. Também se pode fazer essa atribuição com a manipulação direta de ponteiros para a estrutura, mas essa forma de trabalho não será tratada neste capítulo. Os exemplos abaixo mostram o tratamento de campos individuais de uma estrutura.

```
emp.salBase = listaemp[j].salBase;
strcpy (emp.nome, listaemp[j].nome);
```

Vetores e estruturas heterogêneas

A manipulação de vetores de estruturas heterogêneas deve ter ficado evidente a partir dos exemplos anteriores. Isso significa que um determinado campo de uma dada estrutura, que componha um vetor, é acessada primeiro identificando a sua posição no vetor (seu índice). Depois o campo específico é acessado como se acessa um campo de uma estrutura qualquer. Por exemplo, o comando `listaemp[i].descontos=0` atribui o valor zero ao campo "descontos" da posição i do vetor "listaemp".

Entretanto, é possível também que um campo da estrutura seja um vetor. Nesse caso o índice fica associado ao campo da estrutura. Suponha, por exemplo, que temos uma estrutura *diario*, que armazena as notas dos alunos de uma determinada disciplina. Outros campos nessa estrutura envolveriam média da sala, nome da disciplina e do professor e ano de oferecimento.

O trecho de código da figura 7.4 apresenta como essa estrutura seria declarada e manipulada. Observem que na declaração da estrutura fazemos a declaração de campos como sendo vetores (*notas*) e também de um vetor para armazenar as diversas disciplinas (*curso*).

Como a variável *curso* é uma estrutura do tipo vetor, temos que especificar qual de seus elementos queremos acessar (qual disciplina do curso) usando um índice (dado pela variável i no exemplo). Caso o campo acessado também seja um vetor (*notas*), então temos que indicar também qual nota estamos referenciando, o que é feito por um segundo índice (variável j no exemplo).

```
1   struct diario { // define a estrutura chamada "diario"
2     char nomedisc[25];     // com os campos aqui listados
3     int coddisc;
4     char nomeprof[50];
5     float notas[100];
6     float media;
7     int ano;
8   } disciplina, curso[50]; // variável "curso" armazena registros
9                            // das disciplinas do curso
10
11   main()
12   {int i, j;
13   // código apagado
14     for (i=0; i<totdisc; i++)  // laço para todas disciplinas
15       for (j=0; j<n_alun; j++)  // laço para notas da disciplina i
16       { scanf("%f",curso[i].notas[j]);
17         if (curso[i].coddisc == disciplina.coddisc)
18             disciplina.notas[j] = curso[i].notas[j];
19       }
20      // mais código apagado
21   }
```

Figura 7.4: Manipulação de campos com vetores

Funções e estruturas heterogêneas

O uso de estruturas heterogêneas em funções segue o mesmo padrão examinado para os tipos básicos de dados. Ao se definir que um (ou mais) dos parâmetros será uma estrutura heterogênea, o compilador instrui ao processador que reserve os espaços necessários para guardar todos os campos da estrutura. No caso de ser um ponteiro para a estrutura o que ocorre é a passagem por referência do endereço para o início dessa estrutura.

Assim, ao usarmos estruturas heterogêneas como parâmetros em funções temos que ter os mesmos cuidados examinados no capítulo anterior, principalmente no que diz respeito à passagem por valor ou por endereços e os efeitos que isso representa no restante do programa. Vejamos alguns exemplos:

```
salario = calc_salario(emp.salBase, emp.horastrab); /* calcula o
             salário usando os parâmetros salBase e horastrab */

insere_empr(listaemp[pos], emp); /* insere dados de um empregado
      (emp) na posição "pos" do vetor de empregados (listaemp) */
```

7.2.3 Estruturas dentro de estruturas

No exemplo dado para a ficha funcional existe um campo para data de nascimento da pessoa. Lá assumiu-se que essa data seria uma cadeia de caracteres, o que é válido mas complica bastante a sua manipulação. Isso ocorre pois temos que primeiro "separar" os valores de dia, mês e ano, antes de fazer qualquer outra operação. Entretanto, a data de nascimento poderia ser uma segunda estrutura, com os campos dia, mês e ano, em que dia e ano seriam valores inteiros e mês seria uma cadeia de caracteres, como em:

```
typedef struct dias {
   int dia, ano;
   char mes[5]; } Data;
```

Com isso, o campo `datanasc` passaria a ser:

```
Data datanasc;
```

Para acessar os valores da data de nascimento é necessário então identificar o campo externo (`datanasc`) e também o interno (`ano` por exemplo). Isso é feito em comandos como:

```
if (emp.datanasc.ano > 2000)
    do_something;

scanf("%s", emp.datanasc.mes);

listaemp[j].datanasc.dia = emp.datanasc.dia;
```

É importante ressaltar que ao se definir a estrutura separada para armazenar datas, podemos utilizá-la para diversas funcionalidades dentro do mesmo programa. Isso ocorre, por exemplo, para registrar dias de falta de um empregado, datas de início e fim de férias, etc.

7.2.4 Estruturas heterogêneas especiais

As estruturas examinadas até agora se limitam a agrupar dados sobre um elemento sob um único nome simbólico. Existem, entretanto, estruturas que permitem a organização de conjuntos de dados que tenham uma maior relação semântica entre si. Exemplos de relações semânticas incluem relações de ordem

e precedência, entre outras. Em estruturas que possuam relações semânticas as informações contidas ficam divididas em duas partes: uma com as informações particulares de cada elemento e outra com informações sobre as relações de ordem da estrutura formada.

Estruturas desse tipo são essenciais nas aplicações que envolvam enormes quantidades de dados. Por exemplo, numa empresa com muitos empregados (vários milhares) o sistema com o registro dos mesmos deve apresentar alguma característica que permita facilitar a busca pelos dados de um empregado específico. Assim, uma estrutura para armazenar esses registros funcionais deve mantê-los organizados em algo parecido com uma lista ordenada. Os componentes dessa lista devem conter os mesmos elementos vistos na figura 7.2 para armazenar os dados dos funcionários. Além daqueles campos, devem ser acrescentados campos para indicar qual o próximo funcionário na lista. Esse tipo de informação é o que permite o tratamento dos membros da lista e de sua manipulação.

Quando criamos estruturas com funcionalidades semânticas, como listas ordenadas, por exemplo, temos também que criar operações específicas sobre a estrutura. As operações de manipulação incluem a inserção de um novo elemento, a remoção de um elemento e a busca para verificar se um dado elemento está contido na lista. A combinação entre os operadores sobre a estrutura e a estrutura propriamente dita gera o que chamamos de **Tipos Abstratos de Dados** (TAD ou ADT, da sigla em inglês), que são uma das peças fundamentais dentro do conceito de POO (programação orientada a objetos).

EXEMPLO

Considere uma estrutura que armazene os nomes de frutas, além de seu peso médio, sabor e côr. Uma possível estrutura seria:

```
struct fruta {
      char nome[20];
      int sabor; // 1=doce, 2=azedo, 3=amargo, 4=neutro
      int cor; // 1=amarelo, 2=vermelho, 3=verde, ....
      float peso;
} lista[1000];
```

Ao inserirmos alguns tipos de frutas no vetor *lista* pode ocorrer de inserirmos *uva* antes de *goiaba*, e esta antes de *abacate*. Se pretendemos imprimir a lista de frutas com seus pesos médios em ordem alfabética, temos antes que ordenar as frutas dentro do vetor lista.

Uma possível solução é o uso do **algoritmo da seleção** (*selection sort*) apresentado formalmente na Seção 10.2, em que no primeiro passo comparamos o pri-

meiro elemento com os demais da lista e, a cada vez que o primeiro elemento for lexicograficamente posterior ao elemento comparado, deve-se trocá-los de posição. Isso se repete para cada uma das demais posições na lista.

Assim, o algoritmo da seleção permitiria a ordenação de forma bastante simplificada. O problema desse algoritmo é que a cada comparação pode ocorrer a troca dos elementos comparados, o que o torna bastante lento. No caso da lista de frutas a troca de elementos da estrutura tem que envolver a troca de todos os seus elementos. Para determinadas estruturas, envolvendo declarações estáticas de vetores internos, isso deve ser feito de forma individual para cada campo, o que é bastante trabalhoso. Em estruturas simples, como a declarada para as frutas, isso pode ocorrer de forma imediata com a troca direta dos valores de duas posições da lista, como aparece no exemplo a seguir (figura 7.5), que assume que a lista de frutas foi lida em algum outro ponto do programa.

```
1   void ordenaLista (struct fruta *list, int tam)
2   {int i, j, menor;
3    struct fruta aux; // usado para trocar os elementos do vetor
4
5     for (i=0; i<tam-1; i++)
6     { menor = i;
7       for (j=i+1; j<tam; j++)
8       { if (strcmp(list[menor].nome,list[j].nome) > 0)
9             menor = j; // atualiza menor se estiverem invertidos
10       }
11
12       if (menor != i) // atualiza posições das frutas se necessário
13           { aux = list[i];
14             list[i] = list[menor];
15             list[menor] = aux;
16           }
17    }
18  }
```

Figura 7.5: Ordenação de elementos num vetor de estruturas heterogêneas

Essa função parte do princípio de que, inicialmente, a lista com as frutas vem em qualquer ordem e que em cada iteração do primeiro `for` a fruta com menor valor lexicográfico ficará na posição indexada pela variável i.

O que ocorre em cada comando `for` pode ser resumido da seguinte forma, considerando-se que "menor" significa aparecer antes no dicionário:

1. No `for` externo atribui-se inicialmente como menor fruta aquela indicada

pela posição i do vetor;

2. Dentro do `for` interno, busca-se pela menor fruta ainda não classificada, sendo que para isso busca-se entre as frutas localizadas a partir da posição $i + 1$ do vetor;

3. Após identificar a menor fruta, então troca-se a posição da mesma com a da fruta que aparece na posição i da lista;

4. O processo continua até testarem-se todas as posições da lista (na realidade para-se após identificar a penúltima fruta, pois nesse caso a última fruta já estará em sua posição correta na lista).

EXERCÍCIOS

1. Escreva um programa que manipule as notas dos alunos de uma faculdade, usando registros, em que para cada aluno existam as seguintes informações: nome completo, data de nascimento, ano de ingresso, nomes das disciplinas cursadas, quantidade de créditos de cada uma delas, a nota obtida em cada uma delas, além do coeficiente de rendimento do aluno, dado pela fórmula $CR = \frac{\Sigma(Nota_i * NCred_i)}{\Sigma NCred_i}$

2. Escreva um programa para o cálculo do salário de n funcionários, em que as informações sobre os funcionários são nome, número de dependentes, salário por hora e número de horas trabalhadas. Faça os cálculos de desconto de INSS e imposto de renda, usando as seguintes condições:

```
SAL_BRUTO:
   salario/hora * horas trabalhadas + R$ 0,58 por dependente
INSS: 10% se Salario bruto for menor que R$ 5000,00,
        R$ 500,00 caso contrário
SAL_BASE:  SAL_BRUTO - INSS
IR:   isento se SAL_BASE for menor ou igual a R$ 1250,00
      (10% de SAL_BASE - 125,00) se for maior que isso e
                  for menor ou igual a  R$ 2500,00
      (25% de SAL_BASE - 500,00) se for maior que isso
```

   Para cada funcionário informe o nome, salário por hora, horas trabalhadas, salário bruto, desconto de INSS, desconto de imposto de renda e salário líquido.

3. Defina estruturas para armazenar informações sobre livros numa biblioteca. Considere todas as informações cabíveis para os mesmos, tais como empréstimos, reservas, estado de conservação, etc.

4. Escreva funções e procedimentos para manipular dados sobre livros de uma biblioteca. Considere as estruturas definidas no exercício anterior.

5. Defina estruturas para a manipulação de números complexos.

6. Escreva procedimentos para fazer operações sobre números complexos usando as estruturas definidas no exercício anterior.

7.3 Arquivos de dados

Arquivos de dados são instrumentos para se fazer o armazenamento mais permanente de informações. Arquivos ficam armazenados fora da memória, mas apesar disso podem ser (e são) manipulados por programas. Um arquivo é uma entidade lógica que ocupa espaço em dispositivos de armazenamento secundários (discos rígidos, discos flexíveis, discos óticos, fitas magnéticas, etc.) e que ao ser aberto por um programa passa a permitir leituras e escritas sobre seu conteúdo.

As operações que podem ser realizadas sobre um arquivo envolvem, de forma simplificada, sua abertura, seu fechamento e operações de leitura e escrita de conteúdo. A eficiência de um programa que necessite de acesso a arquivos depende, fundamentalmente, da eficiência dessas operações. Aqui ainda não nos preocuparemos em como obter técnicas eficientes de acesso aos dados dos arquivos (o que envolveria o exame de técnicas relativamente sofisticadas para busca e ordenação de informações). Assim, nas próximas páginas examinaremos as formas mais simples de leitura, escrita e demais operações sobre arquivos.

7.3.1 Leitura e escrita

As operações de leitura e escrita são em quase tudo semelhantes às de leitura e escrita examinadas até agora (teclado e vídeo). Uma diferença importante é de que um arquivo possui final, isto é, as informações nele armazenadas são finitas, enquanto a entrada de dados pelo teclado pode prosseguir indefinidamente.

Outra diferença está na forma de acesso aos dados. Enquanto no teclado os dados são obrigatoriamente lidos em sequência, num arquivo o acesso aos dados pode ocorrer de forma sequencial, direta ou indexada. Essa variedade de formas

de acesso surgiu porque o custo (em termos de tempo) de acesso aos arquivos armazenados em dispositivos secundários de memória é extremamente alto em relação aos tempos de acesso à memória e aos registradores da CPU (na prática falamos de tempos na ordem de nanossegundos para registradores, dezenas de nanossegundos para memória e milissegundos para discos). Assim, organizar os dados nos arquivos de forma eficiente passou a ser um problema importante em computação, gerando muitas pesquisas na área de organização e recuperação de informações.

Exceto pelas diferenças mencionadas no paragrafo anterior, a leitura ou escrita de dados em arquivos ocorre de forma igual à de teclado ou vídeo. Na prática, em algumas linguagens usa-se exatamente os mesmos comandos para as operações de entrada e saída de dados, independente do dispositivo usado. Claro que nessas situações deve aparecer no próprio comando alguma forma de identificação do destino ou origem da operação, o que se chama direcionamento de entrada/saída. Alguns sistemas operacionais ainda permitem um redirecionamento dessa operação, permitindo por exemplo que se leia dados de um arquivo embora eles fossem esperados a partir do teclado. Isso, entretanto, está fora dos objetivos deste capítulo.

7.3.2 Abertura

A operação de abertura de um arquivo resulta em uma série de atividades a serem cumpridas pelo sistema operacional do computador que estiver executando o programa. Cronologicamente essas atividades incluem:

1. Busca no sistema de arquivos do computador pelo arquivo que foi solicitado, sendo que se a operação for de leitura é obrigatória a existência prévia do arquivo, ocorrendo falha de execução caso isso não ocorra, e se for de escrita ainda deve-se levar em consideração o tipo de acesso a ser feito (escrita, sobrescrita ou anexação);

2. Registro do arquivo aberto na memória do computador, trazendo-o parcialmente para espaços da memória, caso o arquivo já exista, ou apenas reservando espaço na memória em caso contrário (neste caso apenas se for aberto para escrita);

3. Notificação ao programa sobre a abertura do arquivo e criação de um marcador para o início desse arquivo (esse marcador será usado pelo programa para poder "caminhar" através do conteúdo do arquivo).

Ao final dessas atividades, portanto, o programa recebe um marcador (tipicamente um ponteiro) para o início do arquivo. Esse marcador deve ser usado então para cumprir as atividades de leitura e/ou escrita sobre o arquivo, sendo atualizado a cada comando de leitura/escrita realizado, de forma a ficar sempre posicionado sobre o próximo *byte* a ser acessado no arquivo.

7.3.3 Fechamento

No fechamento de um arquivo o sistema operacional acaba realizando mais uma vez uma série de tarefas, dentre as quais a atualização final dos dados do arquivo no disco. Dentre essas atividades a mais importante é a liberação do espaço que o arquivo estava ocupando na memória. Uma observação importante é de que, após fechado, um arquivo apenas pode ser acessado novamente pelo programa caso seja aberto mais uma vez. Assim, podemos definir o escopo de um arquivo como sendo o período entre sua abertura e o seu fechamento pelo programa.

7.3.4 Fechamento

No fechamento de um arquivo o sistema operacional acaba realizando mais uma vez uma série de tarefas, dentre as quais a atualização final dos dados do arquivo no disco. Dentre essas atividades a mais importante é a liberação do espaço que o arquivo estava ocupando na memória. Uma observação importante é de que, após fechado, um arquivo apenas pode ser acessado novamente pelo programa caso seja aberto mais uma vez. Assim, podemos definir o escopo de um arquivo como sendo o período entre sua abertura e o seu fechamento pelo programa.

7.3.5 Arquivos em C

Um arquivo é manipulado em C através de um ponteiro para o seu conteúdo. Embora isso possa parecer estranho, acaba por facilitar bastante o tratamento de arquivos ao trata-los como uma sequência de endereços em disco.

A declaração de um ponteiro para um arquivo é feita declarando-se um ponteiro do tipo FILE, como visto logo abaixo. O conteúdo desse ponteiro é um endereço na memória correspondente ao local em que está armazenada a próxima informação a ser lida ou escrita no arquivo.

```
FILE  *pont_arq;
```

As manipulações sobre os arquivos são realizadas por funções bastante semelhantes às que examinamos para leitura e escrita padrão (teclado e monitor). As diferenças que surgem são a inclusão do ponteiro direcionador como parâmetro dessas funções e o acréscimo da letra 'f' antes de seus nomes, como em *fputs* em

vez de *puts* por exemplo. Os principais comandos para manipulação de arquivos aparecem na Tabela 7.1 a seguir:

Tabela 7.1: Tabela de funções para manipulação de arquivos, sempre considerando fp como um ponteiro para tipo $FILE$

COMANDO		DESCRIÇÃO
fp = fopen(nome, op)	$\longrightarrow$	abre um arquivo chamado *nome*, para realizar operações indicadas por *op*, retornando o ponteiro *fp* como marcador do arquivo, caso a operação de abertura seja bem sucedida. O tipo *op* pode ser, entre outros, `"r"` para leitura, `"w"` para escrita, `"a"` para anexar
fclose(fp)	$\longrightarrow$	fecha o arquivo apontado por *fp*
fputs(texto, fp)	$\longrightarrow$	equivalente ao *puts*, porém operando sobre o arquivo marcado por *fp*
fscanf(fp, fmt, lista)	$\longrightarrow$	armazena, nas variáveis pertencentes a *lista*, segundo a formatação apresentada por *fmt*, os valores lidos do arquivo apontado por *fp*
fgets(buf, quant, fp)	$\longrightarrow$	copia *quant* bytes para a cadeia de caracteres *buf*
fprintf(fp, fmt, lista)	$\longrightarrow$	grava os conteúdos das variáveis em *lista*, segundo a formatação apresentada por *fmt*, no arquivo apontado por *fp*
feof(fp)	$\longrightarrow$	verifica se chegou ao final do arquivo apontado por *fp*, isto é, passou seu último caracter, retornando 1 se for verdade ou 0 se isso for falso
fseek(fp, quant, origem)	$\longrightarrow$	movimenta o ponteiro *fp* *quant* bytes a partir da posição definida por *origem*, que pode ser a primeira do arquivo (`SEEK_SET`), a última (`SEEK_END`) ou a atual (`SEEK_CUR`)
ftell(fp)	$\longrightarrow$	retorna a posição atualmente apontada por fp

O uso destes comandos é feito de forma similar ao que se faz com os comandos para entrada e saída padrão. O próximo exemplo ilustra claramente que o uso de arquivos dentro de um programa é algo bastante simples de ser feito, bastando para isso que se siga os procedimentos naturais de especificar um bom algoritmo e, a partir dele, fazer a implementação usando os comandos adequados em cada situação.

Uma observação importante a fazer é que na abertura de um arquivo o comando *fopen* pode resultar em erro, quando a abertura for para leitura e o arquivo não existir. Nesse caso o valor de retorno da função *fopen* é NULL. Esse valor de retorno permite que o programa faça o tratamento de erro de abertura do arquivo, como apresentado através do comando a seguir:

```
if (fp = fopen(arquivo,"r") == NULL)
    { puts ("Erro na abertura do arquivo");
      exit(0);
    }
```

Exemplo

Um problema comum em computação é o da conversão de padrões entre dois códigos pré-determinados, em que dados são lidos a partir de um arquivo e escritos em outro após aplicar as transformações necessárias. Considere então que um programa deve ler uma tabela em que cada linha contém os seguintes dados:

```
inteiro1 string1 inteiro2 string2 double1 double2 double3
```

Como resultado de sua execução o programa deve gerar um novo arquivo de dados, em que cada linha conterá:

```
(inteiro1 + inteiro2) string1 (double1 + double2)/double3 string2
```

Um possível algoritmo para esse programa aparece na figura 7.6. Nele o que temos de diferente em relação aos programas apresentados até aqui é exatamente o uso explícito de arquivos, tanto para a leitura de dados quanto para a escrita dos resultados da transformação.

Na prática isso implica em abrir simultaneamente um arquivo para leitura das informações iniciais e outro para a escrita dos resultados aplicados naquelas informações. Na sequência devem ser lidas as informações do primeiro arquivo, linha por linha, realizadas as operações, que devem ser escritas no segundo arquivo. Tanto no algoritmo apresentado, como em sua codificação, fizemos a leitura dos valores separadamente. Isso, obviamente, pode ser modificado para a leitura imediata de vários valores contidos numa linha.

Deve-se observar aqui que, pela definição de como funcionam as funções para a manipulação de um arquivo, seus marcadores são implicitamente atualizados a cada operação sobre o arquivo. É isso que permite que o marcador sobre o arquivo de entrada parta do início do arquivo e chegue até o seu final, quando

```
DECLARE inteiros a, b, c
DECLARE reais x, y, w, z
DECLARE strings n1, n2
DECLARE ponteiros de arquivos arq1, arq2

  ABRIR arquivo de entrada, iniciando ponteiro arq1
  ABRIR arquivo de saída, iniciando ponteiro arq2

  ENQUANTO arq1 não estiver no final do arquivo FAÇA
    LER um inteiro a partir de arq1 e armazene na variável a
    LER um string a partir de arq1 e armazene na variável n1
    LER um inteiro a partir de arq1 e armazene na variável b
    LER um string a partir de arq1 e armazene na variável n2
    LER um real a partir de arq1 e armazene na variável x
    LER um real a partir de arq1 e armazene na variável y
    LER um real a partir de arq1 e armazene na variável w
      FAÇA c = a + b
      FAÇA z = (x + y) / w
    ESCREVER a partir de arq2 os valores de c, n1, z e n2
  FIM ENQUANTO

  FECHAR arquivo de entrada, liberando arq1
  FECHAR arquivo de saída, liberando arq2
```

Figura 7.6: Algoritmo de conversão de dados

isso for necessário. Esse algoritmo leva ao programa da figura 7.7.

Nesse programa mantivemos a estrutura das operações sobre arquivos exatamente como descritas no algoritmo. Fizemos isso apenas para ilustrar como usar os comandos de leitura e escrita sobre arquivos. Num programa real poderíamos agrupar várias operações de leitura (ou escrita quando fosse o caso) em um único comando em C. Um exemplo disso seria a substituição das três leituras de números reais por um único *fscanf*, como o que aparece a seguir:

```
fscanf (arq1, "%lf %lf %lf", &x, &y, &w);
```

```
#include "stdio.h"

main()
{ int a, b, c;
  char n1[20], n2[20];
  double x, y, w, z;
  FILE *arq1, *arq2;

    arq1 = fopen("arquivo.in","r");
    arq2 = fopen("arquivo.out","w");
    fscanf(arq1,"%d",&a);
    while (! feof(arq1))
    { fscanf(arq1,"%s",n1);     fscanf(arq1,"%d",&b);
      fscanf(arq1,"%s",n2);     fscanf(arq1,"%lf",&x);
      fscanf(arq1,"%lf",&y);    fscanf(arq1,"%lf",&w);
      c = a+b;
      z = (x+y)/w;
      fprintf(arq2,"%d %s %lf %s\n",c,n1,z,n2);
      fscanf(arq1,"%d",&a);
    }
    fclose(arq1);
    fclose(arq2);
}
```

Figura 7.7: Código C para o programa de conversão de dados

EXERCÍCIOS

1. Reescreva o programa para o exercício 3 do final do capítulo 2, considerando que os dados dos alunos e disciplinas serão lidos de arquivos.

2. Faça o mesmo para os exercícios 3, 4 e 6 do final do capítulo 6.

3. Faça o mesmo para os exercícios 1, 2 e 4 do final da seção 7.2 deste capítulo.

Capítulo 8

Estruturas dinâmicas

Durante nosso estudo sobre tipos de dados vimos que o tipo de dados ponteiro, que representa endereços na memória (seção 5.4), permitiria a construção de tipos de dados dinâmicos. Dados dinâmicos são aqueles em que os espaços ocupados na memória são criados ou destruídos conforme se tornam necessários ou não. Esta característica faz com que o acesso ao conteúdo de um dado dinâmico tenha que ser feito por meio de seu endereço, pois é impraticável criar tabelas de símbolos para variáveis tão voláteis. Ao longo deste capítulo examinaremos como as estruturas dinâmicas são criadas, para que servem e de que modo podem ser manipuladas pelo programa. Examinaremos também como tudo isso é feito na linguagem C.

Antes de passarmos a esse estudo devemos salientar que nos capítulos seguintes continuaremos a aplicar estruturas dinâmicas, porém sob a ótica de tipos abstratos de dados. Separamos a aplicação de ponteiros na construção de estruturas dinâmicas (aqui) do estudo de tipos abstratos de dados (próximos capítulos) pois esses últimos são corpos completos de estudo, tendo sua implementação independente do uso de estruturas dinâmicas ou de estruturas estáticas (vetores e matrizes).

8.1 Estruturas dinâmicas de dados

Como já deve estar subentendido, as estruturas dinâmicas são estruturas de dados construídas a partir da ideia de que o espaço por elas ocupado não será alocado pela declaração de variáveis, durante a inicialização do programa ou subprograma. Isso implica em alocar, quando necessário, apenas as posições de memória que serão utilizadas naquele momento. O acesso a essas posições é feito por meio de ponteiros de endereços.

O uso de ponteiros para marcar posições alocadas dinamicamente durante a

execução do programa demanda, na prática, a criação de estruturas heterogêneas para a sua manipulação. As estruturas criadas devem ter duas partes, ou no mínimo dois campos:

- uma parte em que se armazena o conteúdo específico da informação, e
- outra parte em que se armazena endereços para outras posições de memória que tenham relação com a informação armazenada na primeira parte.

Isso é representado esquematicamente na figura 8.1, em que a parte superior de cada elemento diz respeito ao dado armazenado (`DADOS A` e `DADOS B`). Já a parte inferior diz respeito ao ponteiro, que apontará para outro dado do mesmo tipo.

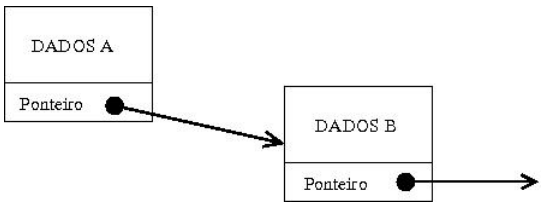

Figura 8.1: Diagrama conceitual de uma estrutura dinâmica

Um detalhe que falta no diagrama da figura 8.1 é como se chega ao trecho de memória em que está a estrutura que armazena `DADOS A`. Isso é feito por uma variável que armazena apenas o endereço dessa estrutura. Assim, tem-se primeiro um ponteiro para a estrutura contendo `DADOS A` e esta estrutura tem um ponteiro para a estrutura com `DADOS B`. Esta outra estrutura, por sua vez, também tem um ponteiro que pode apontar para outra estrutura do mesmo tipo ou marcar o final das estruturas alocadas dinamicamente, apontando para um endereço nulo.

Desse modo, para se criar e manipular uma estrutura dinâmica é preciso primeiro definir o que essa estrutura irá armazenar. Depois é preciso declarar variáveis do tipo ponteiro para o tipo da estrutura definida. Isso diferencia as estruturas dinâmicas dos ponteiros simples, mesmo que apontem para um espaço alocado dinamicamente. A manipulação dos conteúdos de estruturas dinâmicas ocorrerá, portanto, por meio do acesso e manipulação de ponteiros. **Recomenda-se então uma breve revisão do conteúdo da seção 5.4 antes de continuar o estudo deste capítulo**.

8.1.1 Algumas aplicações de estruturas dinâmicas

Estruturas dinâmicas são aplicadas numa variedade impressionante de sistemas computacionais. Na prática, sempre que precisarmos que o espaço ocupado por

nossos dados possa variar de acordo com a necessidade momentânea do usuário, temos uma aplicação imediata de estrutura dinâmica. Isso é verdade num grande número de situações em que se exija o uso das estruturas heterogêneas vistas no capítulo anterior. Vejamos algumas situações em que isso ocorre:

Bancos de dados

Bancos de dados formam um conjunto de aplicações em que tipicamente não se pode determinar exatamente o volume de informações a serem armazenadas. Além disso as operações típicas num banco de dados envolvem a busca por informações a partir de diferentes contextos e a inserção/remoção de informações, sempre de forma eficiente. Uma maneira de se fazer isso é permitir que as relações entre as várias informações sejam dinamicamente estabelecidas. Tudo isso implica no uso de estruturas dinâmicas, que nesses casos envolvem árvores B, árvores k-dimensionais, etc.

Matrizes esparsas

Muitos problemas numéricos (vindos de várias áreas do conhecimento) envolvem a manipulação de matrizes. Em muitos casos tais matrizes são muito grandes (até mesmo com milhões de elementos), porém tendo grande parte desses elementos com valor zero. Matrizes com tais características são chamadas de matrizes esparsas, pois os valores diferentes de zero ficam esparsos nas linhas e colunas da matriz. Uma estrutura estática para esse tipo de matriz implicaria num grande desperdício de memória, pois se alocaria espaço para todos os elementos, independente de serem zeros ou não. Uma estrutura dinâmica permite que se armazene apenas os valores diferentes de zero, economizando bastante memória portanto.

Listas e árvores

Listas e árvores são estruturas especiais que procuram organizar os dados de um problema de forma a facilitar sua manipulação ou ainda criar uma ordem específica entre eles. Aplicações de listas e árvores envolvem a criação de filas de elementos, cadastros ordenados, bancos de dados rudimentares, etc. Nos próximos capítulos examinaremos com detalhes o uso de estruturas dinâmicas em listas e em árvores, verificando tanto porque são usadas como quais são as vantagens em sua implementação nessa forma.

NOTA - Em algumas situações pode ocorrer de se declarar uma variável estática como sendo de um tipo dinâmico. Isso ocorre pois poderemos ter uma variável com espaço reservado de modo permanente armazenando o endereço inicial de alguma estrutura dinâmica mais complexa, como uma lista por exemplo. Devemos lembrar, neste caso, que a variável declarada não será dinâmica, mesmo contendo um ponteiro para espaços alocados dinamicamente.

8.2 Operações básicas em estruturas dinâmicas

Já vimos que uma variável de um tipo de dados dinâmico é composta por duas partes, uma com o conteúdo da variável e outra com o endereço do próximo dado daquele tipo. Assim, as operações sobre variáveis dinâmicas envolvem:

1. Declaração do tipo de dados dinâmico e de sua estrutura dinâmica;
2. declaração de variáveis ponteiro para a estrutura;
3. alocação de espaço na memória para armazenar a estrutura;
4. associação de uma variável ponteiro para o endereço do espaço alocado;
5. manipulação do conteúdo do espaço alocado;
6. associação entre variáveis dinâmicas;
7. liberação do espaço alocado na memória.

Examinaremos agora cada uma dessas operações, sem relacioná-las inicialmente a alguma linguagem.

Declaração da estrutura

Como estruturas dinâmicas são compostas por duas partes (dados e ponteiros para outros dados) fica evidente que são estruturas heterogêneas. Sendo assim, a declaração de uma estrutura dinâmica segue os mesmos procedimentos de estruturas heterogêneas vistos anteriormente.

A parte referente aos dados pode ter qualquer forma, desde um tipo simples até um tipo heterogêneo. A parte referente ao ponteiro deve ser composta por um ou mais ponteiros para variáveis do tipo dinâmico sendo declarado. É esse ponteiro que permite, na prática, a criação de estruturas que podem ser alteradas dinamicamente, aumentando ou diminuindo o espaço ocupado por meio da alocação de memória em endereços que ficam associados a esses ponteiros.

Declaração de ponteiro

Assim como nas estruturas heterogêneas vistas no capítulo anterior, a declaração da estrutura significa apenas especificar quantos bytes serão necessários para armazena-la e quais são os tipos dos campos que a compõe. Para usar um elemento do tipo dessa estrutura é preciso declarar variáveis desse tipo. No caso de estruturas dinâmicas as variáveis declaradas são ponteiros que armazenam o endereço alocado na memória em que os dados efetivos estarão armazenados.

Deve-se lembrar aqui ainda que como se tratam de variáveis dinâmicas, a sua existência na memória não tem a mesma vida, isto é, escopo, das demais variáveis. Esses endereços (e seus conteúdos) apenas fazem sentido, isto é, existem, entre sua alocação e a liberação do espaço na memória.

Alocação de espaço

Diferente das variáveis estáticas, declaradas no início de um programa ou de um subprograma, a declaração de variáveis dinâmicas significa apenas definir um ponteiro que armazenará o endereço de uma variável daquele tipo. Para obtermos efetivamente os bytes a serem ocupados por essa variável na memória é preciso fazer uma operação de alocação de memória.

As linguagens que permitem estruturas dinâmicas possuem comandos específicos para a alocação de espaço. Essa alocação normalmente é feita definindo-se o tamanho, em bytes, que uma unidade daquela estrutura ocupa na memória e quantas unidades dessa estrutura serão alocadas naquele momento. O computador reserva então a quantidade de bytes correspondente e retorna para o programa o endereço do primeiro byte alocado.

Associação do ponteiro ao endereço

Após alocar um conjunto de bytes na memória, temos que associar-lhe um símbolo para seu tratamento pelo programa. Esse símbolo é o nome de alguma variável ponteiro para a estrutura alocada. Isso pode ser feito por uma variável isolada ou com um ponteiro interno a alguma variável do tipo alocado. Veremos mais adiante como isso é feito de forma prática.

Manipulação do conteúdo

O conteúdo de variáveis alocadas dinamicamente é manipulado por meio de ponteiros para seus endereços. Assim, o programador deve tomar o cuidado de utilizar ponteiros de forma correta, evitando que se altere espaços indevidos na memória, principalmente espaços que apontam para outros elementos da estrutura dinâmica.

Manipulação de variáveis dinâmicas

Variáveis dinâmicas são manipuladas de modo equivalente à manipulação de variáveis estáticas, com a exceção do acesso aos seus dados ser feito usando ponteiros. Então, a atribuição de valor a um campo de uma variável dinâmicas deve ser feita identificando o endereço da variável (ponteiro) e qual o campo a receber a atribuição.

Entretanto, uma forma importante de manipulação envolve os campos de endereço. Na prática isso significa atribuir ao campo de endereço o endereço de uma segunda estrutura dinâmica, permitindo estabelecer uma relação entre as duas estruturas. Isso permite a criação de listas, entre outras estruturas de programação, usando ponteiros para estabelecer relações definidas a partir de determinados valores dos dados armazenados.

Por exemplo, se o conteúdo armazenado no endereço $endX$ preceder logicamente o conteúdo armazenado no endereço $endY$, temos que fazer com que o ponteiro da variável armazenada em $endX$ aponte para $endY$. Quando uma estrutura dinâmica usa associações de ordem, esse procedimento de fazer com que um elemento da estrutura aponte para o seguinte prossegue até que se chegue ao último elemento da estrutura. Este último elemento irá, então, apontar para um endereço nulo, marcando o final da estrutura.

Liberação do espaço

A grande vantagem do uso de estruturas dinâmicas é permitir que os espaços da memória sejam ocupados apenas enquanto estritamente necessários. Assim, é preciso que se tenha uma operação para liberação dos espaços que foram alocados tão logo não exista mais uso para os dados neles armazenados.

Toda linguagem que habilite a alocação de memória também fornece mecanismos próprios para a liberação de espaços alocados. O programador deve apenas tomar cuidado para liberar os espaços na quantidade correta e a partir de um endereço inicial correto. Deve ainda lembrar que um espaço liberado não deve ser acessado novamente, a menos que seja realocado com outro conteúdo.

8.3 Tipos de dados dinâmicos em C

A linguagem C habilita a implementação de tipos de dados dinâmicos com o uso de ponteiros e estruturas heterogêneas (*struct*). Assim, basta declarar uma estrutura em que um de seus campos seja um ponteiro para essa própria estrutura. Descrevemos a seguir como cada uma das operações indicadas na seção anterior são implementadas em C.

8.3.1 Declarando uma estrutura dinâmica

As operações de declaração da estrutura e de um ponteiro para ela são feitas de forma bastante simplificada em C. Genericamente temos:

```
typedef struct UmaEstruturaDinamica {
     tipo1  campo1;
     tipo2  campo2;
      ....
     struct UmaEstruturaDinamica  *ponteiro;
} Dinamica;
```

Nesta declaração temos inicialmente a definição dos elementos da estrutura dinâmica, redefinida como sendo do tipo `Dinamica`, contendo vários campos (de qualquer tipo) e um campo contendo um ponteiro para esse tipo de estrutura. É esse campo que permite a construção de estruturas dinâmicas complexas.

Depois da definição da estrutura temos que declarar variáveis dessa estrutura. A seguir se pode ver na primeira linha a declaração de uma variável que é na realidade um ponteiro para uma estrutura do tipo definido e linha seguinte a declaração de uma variável estática para a mesma estrutura.

```
Dinamica  *variavel_ponteiro_desse_tipo;
Dinamica  uma_variavel_simples_desse_tipo;
```

Embora pareça contraditório, o uso de uma variável estática para dados dinâmicos pode ser interessante para o armazenamento de componentes permanentes de estruturas de dados especiais, como por exemplo o início e final de uma fila, início de listas encadeadas, etc. A diferença entre as duas formas está no conteúdo de cada uma. Na forma estática a variável declarada contém todos os campos da estrutura `Dinamica`, incluindo o ponteiro para uma estrutura desse tipo. Já o ponteiro declarado não contém os demais dados da estrutura, contendo apenas o endereço de uma variável do tipo da estrutura, ou seja, é apenas um ponteiro para uma variável, como qualquer outro ponteiro.

8.3.2 Alocação e associação de endereços

Quando se declara um ponteiro para uma estrutura dinâmica ainda não temos espaço na memória para armazenar o conteúdo dessa estrutura. A obtenção do espaço ocorre pela alocação de memória, o que é feito em C com o uso das funções *malloc* e *calloc*. As duas funções retornam um ponteiro do tipo `void`, o que significa que é necessário fazer um *casting* explícito para o tipo da estrutura para a qual se faz a alocação.

A atribuição do endereço de retorno das funções `malloc()` e `calloc()` para

uma variável do tipo da estrutura reservada implica na associação entre o espaço alocado e aquela variável. Daí a necessidade do *casting* explícito na atribuição.

As funções para alocação de memória são descritas a seguir.

Função *malloc*()

A função `malloc()` recebe como parâmetro o número de bytes que devem ser alocados na memória. Se não houver espaço disponível retorna o valor NULL. Se a alocação for bem sucedida a função retorna o endereço do primeiro byte reservado na memória. O exemplo a seguir mostra a aplicação desta função, sendo que nele a função `sizeof()` retorna o tamanho da estrutura em bytes.

```
Dinamica *pont;
pont = (Dinamica *)malloc(sizeof(Dinamica));
```

Ao fazer a alocação com esta função não se modifica o conteúdo armazenado na memória antes da alocação. Assim, o conteúdo da memória alocada pode ser visto como lixo do ponto de vista de sua aplicação.

Função *calloc*()

Esta função recebe dois parâmetros, sendo o primeiro deles o número de elementos a serem alocados e o segundo o tamanho (em bytes) de um elemento daquele tipo. Ela retorna o endereço do primeiro byte alocado, sendo que diferentemente da função `malloc`, a função `calloc` "limpa" o conteúdo da memória, colocando zero em todos os bytes alocados. Um exemplo de seu uso aparece a seguir:

```
Dinamica *pont;
pont = (Dinamica *)calloc(quant, sizeof(Dinamica));
```

8.3.3 Manipulação de variáveis dinâmicas

Ao associar um endereço alocado de memória a uma variável dinâmica é possível fazer seu uso no programa. A manipulação de estruturas dinâmicas em linguagem C é feita sempre usando ponteiros e manipulações de `structs`. A principal diferença entre `structs` estáticas e dinâmicas é a forma de acesso aos campos da estrutura.

Enquanto os campos de uma estrutura estática são acessados pela nomeação "`nomestruct.nomecampo`", a sintaxe para se acessar os campos de uma estrutura heterogênea é dada por "`nomestruct->nomecampo`".

A atribuição de conteúdo para uma variável dinâmica pode ser feita de duas formas: endereço de uma variável do tipo da estrutura ou atribuição de cada

campo de forma individual. Isto é bastante semelhante ao que se faz com estruturas heterogêneas estáticas. O trecho de código a seguir ilustra primeiro a atribuição de campos individuais e depois a atribuição pelo endereço de um elemento de mesmo tipo.

```
Dinamica   *pont, *novo, *outro;
pont = (Dinamica *)calloc(quant, sizeof(Dinamica));
  ....
   pont->campo1 = umDadoTipo1; // Atribuindo campos individuais
   pont->campo2 = umDadoTipo2;
   pont->proximo = NULL; // Indica que não há próximo elemento
  ....
   novo = pont; // Atribuindo a estrutura por seu ponteiro
  ....
   novo->proximo = outro; // "novo" e "pont" apontam para "outro"
```

O que precisa ficar claro é que esse processo é semelhante à manipulação de estruturas heterogêneas estáticas. A diferença é o tratamento de ponteiros, tanto como campos da estrutura quanto como endereços apontando para a própria estrutura. Ao longo da seção 8.4 o leitor poderá ver com detalhes o uso dos ponteiros na implementação e manipulação de uma estrutura dinâmica que forma uma lista de dados.

8.3.4 Liberação de memória

A liberação de memória, após o seu uso, é necessária para que as vantagens de economia de espaço com o uso de estruturas dinâmicas realmente apareçam. A linguagem C oferece a função `free()` para que se faça a liberação de espaços previamente alocados. Esta função recebe como parâmetro o endereço do espaço a ser liberado e retorna um valor diferente de zero se a operação for bem sucedida. Esse valor de retorno normalmente é desprezado, sendo útil apenas em situações em que o gerenciamento da memória se torne vital. Na maioria das vezes a função `free()` é chamada de modo isolado, como em:

```
Dinamica   *pont;
pont = (Dinamica *)malloc(sizeof(Dinamica));
  ....
free(pont);
```

Como a função `free()` libera, a cada chamada, a quantidade de memória que foi alocada para aquele ponteiro, então se deve tomar o cuidado de chamar várias vezes a função para a liberação de posições alocadas em momentos diferentes. Isso

implica no uso de estruturas de repetição em que o controle do laço normalmente está atrelado à verificação do endereço apontado, como *"repetir até que o ponteiro seja nulo"*.

8.4 Revisitando a lista de frutas

No capítulo anterior vimos um pequeno exemplo de uso de uma lista de informações sobre frutas. Naquele exemplo usamos uma estrutura estática (um vetor de 1000 posições) para armazenar a lista, quer ela tivesse apenas 2 ou quase 1000 frutas. Veremos aqui como aquela estrutura e programa são modificados para fazer uso de uma estrutura dinâmica.

A primeira modificação envolve a criação de uma nova estrutura para a lista, que terá como campos a estrutura fruta (já conhecida) e um ponteiro para o próximo elemento da lista. Assim, o vetor `lista[1000]` deixa de existir, sendo trocado por um ponteiro para essa lista. Para exercitar aspectos relativos à passagem de parâmetros o ponteiro para a lista será declarado na função *main* e não armazenará conteúdo, sendo apenas um indicador de quem é o primeiro elemento da lista. Isso, como veremos, facilita o trabalho de manipulação da lista dinâmica.

```
struct lista {
  struct fruta no_fruta; // contém dados de uma fruta
  struct lista *prox;    // aponta para próxima fruta da lista
};
```

Como a lista de frutas será criada a partir da alocação de espaço para cada fruta, então é possível gerar a lista inserindo as frutas já de forma ordenada. Se usássemos um vetor, como no capítulo anterior, fazer a inserção já de forma ordenada demandaria muitas trocas de posição dos elementos no vetor. Por exemplo, se tivéssemos que inserir uma fruta no começo da lista teríamos que mover, uma posição adiante, todas as frutas posteriores a ela. Por isso, no caso do uso de um vetor é mais prático primeiro ler todos os elementos e depois ordená-los uma única vez.

Ao usarmos estruturas dinâmicas a ordenação pode ocorrer durante cada inserção. Para isso basta usar o conteúdo do campo `prox`, fazendo com que ao encontrarmos a posição em que o novo elemento deve ser inserido realizemos as seguintes operações: 1) atribuir o endereço do elemento que deve sucede-lo na lista ao campo `prox` do elemento a ser inserido; 2) atribuir o endereço deste elemento para o campo `prox` do elemento que o precederá na lista. Claro que cuidados adicionais são necessários para casos especiais, como ainda não existirem elementos na lista ou o elemento a ser inserido tenha que ser o primeiro ou

último da lista.

O código para inserção é visto na figura 8.2. Como a inserção de um novo elemento na lista, já de forma ordenada, pode ocorrer em qualquer posição, incluindo seu início, a função para inserção deve retornar sempre o endereço do primeiro elemento da lista (pode ser o novo elemento ou o mesmo de antes da chamada). Com isso, a função passa a ser do tipo "`struct lista *`". Esse procedimento deve também ser realizado para as funções que possam inserir ou remover elementos da lista.

Os pontos importantes no código da função `insereLista` são os comandos `if` e o comando `while`. Neste exemplo consideraremos que se a lista estiver vazia, então o ponteiro para ela será nulo. Assim, o primeiro comando `if`, linha 6, verifica se a lista está vazia, inserindo o elemento como sendo o primeiro da lista nesse caso. Caso já exista alguma fruta passa-se ao restante da função, buscando-se a posição exata em que o elemento deve ser inserido.

A busca pela posição em que o elemento deve ser inserido numa lista não vazia é feita no comando `while`, linha 11, cuja condição de parada envolve dois testes. O primeiro verifica se já se chegou ao final da lista (`aux` com valor NULL) e o segundo verifica se o elemento atual da lista precede o novo elemento (usando a função `strcmp`). Para permitir a inserção da nova fruta entre duas outras frutas se utiliza os ponteiros `antes` e `aux` para marcar respectivamente as frutas que antecedem e sucedem a nova fruta na lista.

O comando `if` que aparece na linha 16 vai determinar se a fruta será inserida na primeira posição da lista ou depois disso. No caso de ser a primeira da lista se deve fazer com que a fruta inserida aponte para o antigo início (`inic`) e que esse ponteiro passe a apontar para a nova fruta. Caso a fruta seja inserida após a primeira posição basta fazer o elemento apontado por `antes` apontar para a nova fruta e ela apontar para `aux`. Observe-se que isso funciona também se a nova fruta for a última fruta da lista, pois nesse caso a fruta passará a apontar para NULL (valor de `aux`), tornando-se o novo final da lista.

8.4.1 Completando o programa exemplo

Para finalizar este capítulo, e também este exemplo, apresentamos o restante de seu código. Assumiremos que os dados sobre as frutas são armazenados num arquivo de dados (`pomar.dat`, por ex.) e que o usuário poderá inserir novas frutas, alterar informações já cadastradas ou até remover frutas do cadastro. Qualquer alteração será então salva no arquivo de entrada quando terminar a execução do programa. Para chegarmos até o programa completo precisamos definir primeiro

```
1   struct lista *insereLista(struct lista *inic, struct lista *nova)
2   // *inic tem o endereço do começo da lista
3   // *nova tem o endereço da posição com os dados da nova fruta
4   {struct lista *aux, *antes;  // usados para percorrer a lista
5
6     if (inic == NULL) // lista está vazia?
7        { inic = nova; // faz a nova fruta ser o começo
8          return(inic);
9        }
10    aux = antes = inic;
11    while (aux &&
12           (strcmp(aux->no_fruta.nome,nova->no_fruta.nome) < 0))
13    { antes = aux;         // busca ponto de inserção
14      aux = aux->prox;
15    }
16    if (aux == inic) // pode ser no começo da lista
17        { nova->prox = inic;    inic = nova; }
18      else  // ou não, até mesmo no fim se aux==NULL
19        { nova->prox = aux;    antes->prox = nova; }
20    return(inic);
21  }
```

Figura 8.2: Inserção de novo elemento numa estrutura dinâmica.

o modelo para o tratamento do cadastro, que é apresentado na figura 8.3.

1. O cadastro deve ser uma lista dinâmica ordenada pelo nome das frutas.
2. Os dados sobre as frutas estão guardados num arquivo de texto comum.
3. O programa lerá os dados do arquivo, armazenando-os na lista dinâmica.
4. Após isso o programa deve permitir ao usuário:
   (a) Inserir nova fruta no cadastro
   (b) Procurar por uma fruta no cadastro
   (c) Remover uma fruta do cadastro
   (d) Alterar dados de uma fruta no cadastro
5. Ao terminar o programa deve atualizar o cadastro no arquivo de entrada.

Figura 8.3: Modelo do programa para cadastro de frutas.

Partindo então deste modelo podemos chegar ao programa aplicando a técnica de refinamentos sucessivos no algoritmo. O algoritmo em nível mais alto é apresentado na figura 8.4. Esse algoritmo é claramente uma representação simplificada do modelo, sendo que os refinamentos devem ocorrer sobre cada um dos

procedimentos nele listados.

1. Ler os dados do arquivo de entrada, armazenando-os numa lista ordenada
2. Ler opção do usuário (inserir, procurar, alterar, remover, sair)
3. Enquanto opção for diferente de sair faça:
   Caso opção do usuário seja:
   (a) Inserir, então chame a função `insereLista()`;
   (b) Procurar, então chame a função `procuraFruta()`;
   (c) Remover, então chame a função `removeFruta()`;
   (d) Alterar, então chame a função `alteraCadastro()`;
   Ler opção do usuário (inserir, procurar, alterar, remover, sair)
4. Grave os dados atualizados e termine a execução do programa.

Figura 8.4: Algoritmo em alto nível do programa para cadastro de frutas.

Uma observação bastante importante a ser feita é de que o programa ficará a maior parte do tempo no laço do passo 3. Dentro desse laço o programa executa, a cada iteração, a operação escolhida pelo usuário. Essas operações apresentam uma característica comum, que é de percorrer a lista buscando uma determinada condição (posição da nova inserção ou do elemento a ser exibido, modificado ou removido). Assim, ter um bom procedimento de busca permite tornar o programa mais eficiente, o que é possibilitado mantendo-se as frutas ordenadas.

Do modelo apresentado na Figura 8.3 tem-se que a operação de inserção de uma fruta no cadastro foi descrita na página anterior e seu código C aparece na Figura 8.2. Assim, falta descrever o processo de leitura dos dados, a busca por uma fruta, a remoção de uma fruta e a alteração de dados cadastrais. Esses procedimentos são apresentados a seguir.

Refinamento da leitura do arquivo

A função para leitura de dados do arquivo deve abri-lo, ler seu conteúdo e armazenar esses dados na lista de frutas. Sem detalhar os passos necessários[1], um algoritmo para a leitura é visto na figura 8.5. Os códigos desta função (e também das demais) serão apresentados após a discussão de todos algoritmos.

Refinamento da procura por fruta

Tendo uma lista de frutas ordenada, o algoritmo de busca é bastante simples. Basta percorrer a lista, verificando em cada posição se a fruta nela armazenada

[1]por motivos ligados a espaço e pelo fato de boa parte do código apresentado na seção 8.4.2 ser autoexplicativo, não apresentaremos aqui nenhum algoritmo detalhado

1. Abrir o arquivo de dados para leitura
2. Enquanto não chegar ao final do arquivo faça:
   (a) Leia os dados de uma fruta
   (b) Chame a função de inserção para essa fruta
3. Fechar o arquivo de entrada
4. Guardar (apontar para) o endereço da primeira fruta da lista

Figura 8.5: Algoritmo para leitura dos dados do arquivo de entrada.

é a desejada. Faz-se isso até que se encontre a fruta ou que a fruta da posição atual seja lexicograficamente posterior à fruta procurada, ou ainda se chegue ao final da lista. Esse algoritmo é visto na figura 8.6.

1. Apontar para a primeira fruta
2. Enquanto não chegar ao final da lista e a fruta procurada for diferente da fruta apontada e a fruta procurada for posterior à fruta apontada faça:
   Apontar para a fruta seguinte da lista
3. Se existir fruta apontada e fruta apontada for a desejada
   Então, apresente os dados da fruta apontada
   Senão, insira a fruta no cadastro

Figura 8.6: Algoritmo para procurar uma fruta na lista.

No caso de não se encontrar a fruta procurada podemos adotar duas posturas distintas. Podemos apenas dizer que a fruta não está cadastrada e parar. Ou então podemos acrescentar a fruta à lista, chamando então a função de inserção de uma fruta vista na página 163, que é a forma a ser adotada aqui.

Refinamento da remoção de fruta

A remoção de uma fruta também começa pela busca da fruta a ser removida. Como um elemento será retirado da lista, então é preciso conhecer os endereços da fruta sendo examinada (para saber se é a que será retirada) e da fruta anterior. Isso é necessário pois a fruta anterior deverá, após a remoção, apontar para a fruta que sucedia a que for retirada da lista. Isso cria ainda uma condição extra, que é verificar se a fruta a ser removida não é a primeira da lista. O algoritmo para essa função está apresentado na figura 8.7.

1. Apontar para a primeira fruta
2. Se ela for a fruta a ser retirada
   Então faça a próxima fruta ser o início da lista e remova a fruta desejada, retornando o endereço do novo início da lista
3. Enquanto não chegar ao final da lista e a fruta procurada for diferente da fruta apontada e a fruta procurada for posterior à fruta apontada faça:
   Apontar para a fruta seguinte da lista e guardar endereço da fruta atual
4. Se existir fruta apontada e fruta apontada for igual a desejada
   Então, remova a fruta, atualizando o endereço apontado por sua antecessora
   Senão, indique que a fruta já não existia e retorne o início da lista.

Figura 8.7: Algoritmo para remover uma fruta da lista.

Refinamento da alteração de fruta

Para modificar informações de uma das frutas armazenadas é preciso primeiro localizá-la. Após localizar a posição da fruta, a alteração de seu conteúdo é bastante simples, consistindo na atribuição de um novo valor para um determinado campo. O algoritmo apresentado na Figura 8.8 ilustra todo o procedimento, com destaque para o passo 2, que busca pela fruta na lista. Nessa função assumiremos que a fruta já está na lista. Caso não esteja nenhuma outra ação será realizada.

1. Apontar para a primeira fruta
2. Enquanto não chegar ao final da lista e a fruta procurada for diferente da fruta apontada e a fruta procurada for posterior à fruta apontada faça:
   Apontar para a fruta seguinte da lista
3. Se existir fruta apontada e fruta apontada for igual a desejada
   Então, pergunte qual campo quer alterar e faça a alteração
   Senão, diga que não encontrou a fruta

Figura 8.8: Algoritmo para modificar dados de uma fruta da lista.

Refinamento da escrita do arquivo

A escrita do cadastro no arquivo, após o término de uma sessão de uso do programa, é bastante simples. Basta abrir o arquivo que contém o cadastro e percorrer a lista copiando seus campos para o arquivo. É natural que a ordem da

1. Abrir o arquivo de dados para escrita
2. Apontar para o primeiro elemento da lista de cadastro
3. Enquanto houver fruta no cadastro faça:
   (a) Escreva os dados dessa fruta no arquivo
   (b) Aponte para a próxima fruta do cadastro
   (c) Libere o espaço na memória ocupado pela fruta que foi escrita
4. Fechar o arquivo de saída

Figura 8.9: Algoritmo para escrita dos dados no arquivo de saída.

escrita dos dados deve seguir explicitamente a mesma ordem usada na operação de leitura do arquivo, pois caso contrário aquela função deixaria de funcionar corretamente. Os comandos para realizar a escrita aparecem na Figura 8.9.

8.4.2 O programa de cadastro de frutas completo

As funções descritas podem ser facilmente transformadas em código C, que é apresentado ao longo da Figura 8.10. Começamos com a definição das estruturas a serem usadas, tanto para cada fruta como também para a lista de cadastro das frutas, além dos protótipos das funções a serem usadas.

Deve ser observado que optamos por modularizar o código ao máximo, acrescentando funções para ler dados sobre as frutas, incluindo a conversão entre números e sabores ou cores. Também se optou por fazer o ponteiro para a lista iniciar com valor nulo para indicar lista vazia.

Os protótipos declarados representam as funções dos algoritmos descritos, além de funções para manipulação simples. As primeiras quatro funções são aquelas em que a lista pode ser alterada e, por isso, retornam o endereço do primeiro elemento da lista. Não serão apresentados aqui os códigos para as funções *converteSabor()*, *converteCor()* e *preencheFruta()*, pois são apenas interfaces para conversão de valores ou leitura dos dados de uma fruta a ser inserida.

Deve ser observado na função *main* (Figura 8.10 - parte 2) que o programa ficará bastante tempo no laço contendo o comando *switch*, usado para identificar as ações a serem executadas. Fora do laço o programa apenas faz a inicialização da lista de cadastros, chama a função para leitura do arquivo de entrada, e chama a função para salvar as frutas cadastradas. Adianta-se aqui que os arquivos usados na entrada e saída de dados serão os mesmos, o que significa que numa próxima execução as frutas já serão lidas de forma ordenada.

Na função de leitura do arquivo (Figura 8.10 - parte 3) é importante observar que a cada fruta lida se "monta" um elemento do tipo `struct lista`, que será inserido na lista de frutas cadastradas. Isso ocorre dentro do laço `while` que

```
1   #include "stdio.h"
2   #include "stdlib.h"
3   #include "string.h"
4
5   struct fruta {
6       char nome[20];
7       int sabor; // 1=doce, 2=azedo, 3=amargo, 4=neutro
8       int cor; // 1=amarelo, 2=verde, 3=vermelho, ....
9       double peso;  };
10
11  struct lista {
12      struct fruta no_fruta;
13      struct lista *prox;  };
14
15  struct lista *insereLista(struct lista *, struct lista *);
16  struct lista *removeFruta(struct lista *, char *);
17  struct lista *procuraFruta(struct lista *, char *);
18  struct lista *leArquivo(struct lista *);
19  void alteraCadastro(struct lista *, char *);
20  void salvaLista(struct lista *);
21  char *converteSabor(int);
22  char *converteCor(int);
23  struct lista *preencheFruta(char *);
```

Figura 8.10: Programa de controle de cadastro de frutas (parte 1)

percorre o arquivo até seu final. Já o primeiro laço verifica se o arquivo a ser aberto já existe. Nessa versão se obriga a ter um arquivo inicial de cadastro, mesmo que vazio.

Uma alternativa a esta exigência seria perguntar ao usuário se existem dados já cadastrados, abrindo o arquivo apenas nesse caso. Isso, entretanto, seria mais indicado antes de fazer a chamada para a função de leitura do arquivo, dentro da função *main*.

A função para salvar o cadastro em um arquivo, que é chamada ao final da execução do programa é também relativamente simples. Basicamente o que se faz é reabrir o arquivo usado como entrada de dados e escrever nele a lista de frutas, lembrando de liberar a memória (`free`) a cada fruta salva, como aparece no código apresentado na Figura 8.10 - parte 4. Neste momento algumas observações precisam ser feitas. A primeira é que como o arquivo para escrita é aberto mesmo que não exista, então o nome usado (`"pomar.dat"`) irá criar o arquivo padrão para as próximas execuções do programa.

```
1   void main()
2   {int opcao=0;
3    char umafruta[30];
4    struct lista *cadastro, *novafruta;
5
6     cadastro = NULL;  // indica que a lista está inicialmente vazia
7     cadastro = leArquivo(cadastro); // lê as frutas cadastradas
8     puts("Qual a operação a ser realizada?");
9     puts(" 0 = Sair\n 1 = Inserir fruta\n 2 = Procurar fruta");
10     puts(" 3 = Remover fruta\n 4 = Alterar cadastro de fruta");
11     scanf("%d",&opcao);
12     while (opcao != 0)  // Laço para interação com o programa
13     { switch(opcao)
14       { case 1: puts("Qual fruta a ser inserida?");
15                 scanf("%s",umafruta);
16                 novafruta = preencheFruta(umafruta);
17                 cadastro = insereLista(cadastro, novafruta);  break;
18         case 2: puts("Qual fruta deseja procurar?");
19                 scanf("%s",umafruta);
20                 cadastro = procuraFruta(cadastro, umafruta);  break;
21         case 3: puts("Qual fruta deseja remover?");
22                 scanf("%s",umafruta);
23                 cadastro = removeFruta(cadastro, umafruta);   break;
24         case 4: puts("Qual fruta deseja alterar o cadastro?");
25                 scanf("%s",umafruta);
26                 alteraCadastro(cadastro, umafruta);           break;
27       }
28       // Mostra menu novamente
29       scanf("%d", &opcao);
30     }
31     salvaLista(cadastro);
32   }
```

Figura 8.10: Programa de controle de cadastro de frutas (parte 2)

A segunda observação é que a ordem de escrita dos parâmetros de cada fruta deve seguir exatamente a mesma ordem esperada pela função de leitura. Isso é feito com o comando `fprintf`, dentro do laço que percorre a lista de frutas.

Uma outra observação é a de que como a leitura da lista, para gravação no arquivo, é feita sequencialmente, então o conteúdo do arquivo estará já ordenado. Isso implica que, nas próximas execuções, cada fruta lida será inserida no final da lista. Para otimizar a leitura do arquivo e montagem da lista de frutas poderíamos

```
1   // Função para ler dados de frutas a partir de um arquivo.
2   struct lista *leArquivo(struct lista *head)
3   {struct lista *aux;
4    char nomefruta[30], nomearq[40]="pomar.dat";
5    FILE *fp;
6
7     while (! (fp = fopen(nomearq, "r")))
8     { printf("'%s' não existe! Qual deve ser lido?\n",nomearq);
9       scanf("%s",nomearq);
10    }
11    fscanf(fp,"%s",nomefruta);
12    while(!feof(fp))
13    { aux = malloc(sizeof(struct lista));
14      aux->prox = NULL;
15      strcpy(aux->no_fruta.nome, nomefruta);
16      fscanf(fp,"%d %d %lf",&aux->no_fruta.sabor,
17                  &aux->no_fruta.cor, &aux->no_fruta.peso);
18      head = insereLista(head, aux); // para inserir na lista ....
19      fscanf(fp,"%s",nomefruta);
20    }
21    if (head->prox == NULL)
22        printf("Arquivo '%s' está vazio, nada foi cadastrado.");
23    fclose(fp);
24    return(head);
25  }
```

Figura 8.10: Programa de controle de cadastro de frutas (parte 3)

salvar a lista de modo invertido, mas isso dificulta a escrita do arquivo.

Falta apresentar ainda os códigos para as funções chamadas dentro do laço principal da função *main*. Não será descrito aqui o código de inserção de uma nova fruta, pois o mesmo foi descrito na seção anterior (Figura 8.2, pág. 163). O código para as demais funções é apresentado na sequência.

O primeiro código a ser examinado é o da função para a remoção de uma fruta, visto na Figura 8.10 - parte 5. Como a fruta a ser removida pode ser ou a primeira da lista ou estar depois da primeira, é preciso tomar cuidado para não se perder a parte da lista que vem após ela. No caso dela ser a primeira fruta é preciso atualizar o ponteiro para o início da lista. Se ela não for a primeira é preciso manter um ponteiro para a fruta anterior a ela, que passará a apontar para a fruta seguinte.

Antes disso é preciso localizar a fruta na lista, o que é feito no comando `while`

```
1   // Função para salvar frutas cadastradas no arquivo "pomar.dat"
2   void salvaLista(struct lista *head)
3   {struct lista *aux;
4    FILE *fp;
5
6     aux = head;
7     fp = fopen("pomar.dat","w");  // salva no arquivo padrão
8     while (head != NULL)          // uma fruta por vez
9     { fprintf(fp,"%s %d %d %8.2lf\n",aux->no_fruta.nome,
10                 aux->no_fruta.sabor, aux->no_fruta.cor,
11                 aux->no_fruta.peso);
12      head = aux->prox;
13      free(aux);   // libera a memória ocupada pela fruta salva
14      aux = head;
15    }
16    fclose(fp);
17  }
```

Figura 8.10: Programa de controle de cadastro de frutas (parte 4)

da linha 6. As condições de parada do laço ocorrem se encontrar a fruta ou se ela não puder mais ser encontrada (final da lista ou fruta com nome posterior à procurada). Para identificar a razão da parada se examina se a fruta apontada por `aux` é a procurada com o comando `if (! strcmp(aux->no_fruta.nome, nome))`. Se for verdade remove-se a fruta e atualiza-se a fruta anterior (`prev`) ou o ponteiro para o início da lista caso a fruta retirada fosse a primeira.

Já a função para modificação de cadastro, vista na parte inferior da Figura 8.10 - parte 5, primeiro procura pela fruta a ser modificada no laço `while`. Se a fruta não for encontrada se posta uma mensagem avisando a inexistência da fruta e encerrando a função.

Caso a fruta seja encontrada, se passa para a etapa de modificação, que consiste em perguntar qual parâmetro a ser alterado e realizar esse alteração. A única restrição aqui é que não se pode alterar o nome da fruta, supondo, é claro, que não se erre seu nome quando for inserida.

Finalmente, o código da função para buscar uma fruta no cadastro aparece na Figura 8.10 - parte 6. Nessa função se faz a inserção da fruta procurada caso ela ainda não esteja cadastrada. Isso ocorre com a chamada da função `insereLista` dentro da parte `else` do comando `if`.

A busca pela fruta é feita do mesmo modo já visto nas funções anteriores. Caso a fruta seja encontrada dentro do laço de busca, o que se faz é apresentar suas características, ou seja, nome, peso, sabor e cor.

```
1   // Função para retirar uma fruta do cadastro
2   struct lista *removeFruta(struct lista *head, char *nome)
3   {struct lista *aux = head, *prev = head;
4
5     while (aux->prox && strcmp(aux->no_fruta.nome, nome) < 0)
6     { prev = aux;       // busca pela fruta, mantendo ponteiro
7       aux = aux->prox;  // para a posição anterior
8     }
9     if (! strcmp(aux->no_fruta.nome, nome))  // se encontrou
10        { if (aux != head)     // se estiver no meio da lista
11              prev->prox = aux->prox;
12            else               // se for a primeira da lista
13              head = aux->prox;
14          free(aux);
15        }
16      else     printf("A fruta %s não está no cadastro\n\n", nome);
17    return(head);
18  }
19
20  // Função para modificar dados de uma fruta cadastrada //
21  void alteraCadastro(struct lista *head, char *nome)
22  {struct lista *aux =  head;
23   int troca;
24
25    while (aux->prox && strcmp(aux->no_fruta.nome, nome) < 0)
26        aux = aux->prox;
27    if (strcmp(aux->no_fruta.nome, nome) == 0)
28        { puts("O que se deseja alterar? 1=cor 2=sabor 3=peso");
29          scanf("%d",&troca);
30          switch(troca)
31          { case 1: puts("Qual a cor correta?\n1=amarelo, 2=verde, ...");
32                    scanf("%d",&aux->no_fruta.cor);       break;
33            case 2: puts("Qual o sabor correto?\n1=doce, 2=azedo, ....");
34                    scanf("%d",&aux->no_fruta.sabor);     break;
35            case 3: puts("Qual o peso correto em gramas?");
36                    scanf("%lf",&aux->no_fruta.peso);     break;
37          }
38        }
39      else        printf("Não localizei %s\n\n",nome);
40  }
```

Figura 8.10: Programa de controle de cadastro de frutas (parte 5)

Deve-se observar que a função retorna o ponteiro para o início da lista. Isso é feito pois eventualmente pode ocorrer a inserção de uma nova fruta se ela ainda não estiver no cadastro. Como essa inserção pode ocorrer no início da lista, então é preciso retornar seu endereço, quer tenha sido alterado ou não.

```
1   /* Função para buscar uma fruta no cadastro e, caso não a   /
2   /  encontre, fazer seu cadastro                            */
3   struct lista *procuraFruta(struct lista *head, char *nome)
4   {struct lista *aux;
5
6     aux = head;
7     while (aux->prox && strcmp(aux->no_fruta.nome, nome) < 0)
8        aux = aux->prox;  // Laço de busca pela fruta
9     if (! strcmp(aux->no_fruta.nome, nome)) // Se encontrou ...
10         { printf("Dados de %s:\n",nome);
11           printf("Peso=%8.2f gramas\n",aux->no_fruta.peso);
12           printf("Sabor=%s\n",converteSabor(aux->no_fruta.sabor));
13           printf("Cor=%s\n\n",converteCor(aux->no_fruta.cor));
14         }
15       else        // se não encontrar, faça sua inserção
16         { printf("%s não tem cadastro. Será cadastrada\n\n",nome);
17           head = insereLista(head, preencheFruta(nome));
18         }
19     return(head);
20  }
```

Figura 8.10: Programa de controle de cadastro de frutas (parte 6)

EXERCÍCIOS

1. Refazer os exercícios 1 a 4 do final da seção 7.2, agora usando estruturas dinâmicas.

2. Implementar uma função que receba uma lista genérica, com seus elementos ordenados de forma crescente, e retorne uma segunda lista, agora com os elementos ordenados de forma decrescente.

3. Implemente uma função que receba uma lista genérica e um certo elemento, retornando ou o endereço desse elemento ou o valor NULL caso não o encontre na lista.

4. Considere que uma dada estrutura dinâmica deve armazenar os movimentos

realizados em uma partida de xadrez. Especifique a estrutura dos elementos dessa lista e implemente as funções para inserção das jogadas.

Capítulo 9

Listas, pilhas e filas

No capítulo anterior mencionou-se que com o uso de estruturas heterogêneas é possível criar estruturas de dados mais complexas, chamadas de Tipos Abstratos de Dados (TAD), que é o caso de listas e suas variações. A ideia por trás de um TAD é permitir que não apenas se defina quais informações serão armazenadas mas também os operadores sobre essas informações, ou seja, se define um tipo de dados diferente dos tipos básicos e as operações fundamentais sobre esse novo tipo de dados.

Ao longo deste capítulo examinaremos inicialmente a fundamentação de listas como um tipo de dados, seguindo-se então para uma breve descrição dos vários tipos de listas e de como elas podem ser implementadas computacionalmente. A segunda parte do capítulo contém um exame detalhado de cada tipo de lista, incluindo então os métodos para inserção e remoção de dados na lista.

Antes, porém, é preciso deixar claro que embora a imensa maioria das implementações de listas seja feita com o uso de estruturas dinâmicas, vistas no capítulo 8, as mesmas também podem ser implementadas com estruturas estáticas. Para isso basta usar vetores para armazenar dados da estrutura e indexadores para estabelecer suas relações. Veremos mais adiante um pequeno exemplo de como isso é feito, salientando que no restante do capítulo as implementações serão apresentadas usando estruturas dinâmicas.

9.1 Listas como um Tipo Abstrato de Dados

Listas formam uma classe especial de estruturas para armazenar informações organizadas de modo específico. A ideia básica com uma lista é definir um modelo para organizar os dados que nela serão incluídos de modo a que se tenha controle sobre as relações entre um dado e outro. Assim, o principal aspecto em uma lista é especificar como se estabelecem essas relações, o que leva à definição de

vários tipos diferentes de lista. Além disso, independente do tipo de lista é possível diferenciar sua abordagem quanto à forma de implementação. Essas classificações são vistas a seguir.

9.1.1 Formas de implementação

Do ponto de vista de implementação uma primeira classificação envolve o uso ou não de estruturas dinâmicas. Como já mencionado, uma lista pode ser implementada com estruturas dinâmicas, em que os elementos são alocados na memória quando necessários, usando ponteiros para conectar esses elementos. Estruturas estáticas, por outro lado, envolvem o uso de vetores, com posições já alocadas na memória, conectando os elementos usando indexadores para as posições do vetor.

Uma segunda diferença na forma de implementar uma lista diz respeito a como ela será referenciada no programa. Isso, na prática, significa identificar qual elemento da lista deve ser acessado inicialmente (supondo que acessos aos elementos sejam sequenciais) e como esse acesso ocorre. Como dissemos, listas são formadas por um conjunto de elementos contendo dados e um indexador para o elemento vizinho na lista, sendo que este indexador vai ser usado para passar de um elemento para outro da lista. A questão que resta é como chegar ao elemento "inicial" da lista, usualmente denominado **cabeça da lista**. Isto pode ser feito de duas formas, com impactos distintos sobre operações nas listas. Elas são:

i) **Referência da lista é seu primeiro elemento** - aqui a variável usada no programa para indicar o começo da lista contém também seu primeiro elemento, como visto na Figura 9.1. Isso evita desperdício de uma variável apenas para apontar quem é o primeiro elemento, porém implica em cuidados maiores com as operações de inserção e remoção da lista, além de algum mecanismo para tratar listas vazias.

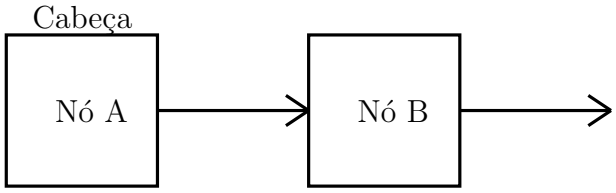

Figura 9.1: Lista com primeiro elemento como cabeça

ii) **Referência da lista aponta para o primeiro elemento** - nesse caso usa-se uma variável, do mesmo tipo da lista, apenas para indexar (apontar para) o elemento que está no começo da lista. Isso implica em se ter uma posição de memória apenas servindo de indexador, porém agora as operações de

remoção, inserção e identificação de lista vazia são facilitadas por não ser necessário trocar o identificador de começo da lista, exceto possivelmente o seu conteúdo.

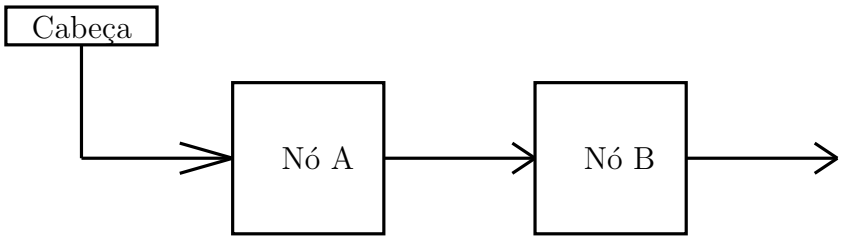

Figura 9.2: Lista com ponteiro externo para seu primeiro elemento

Para listas usando vetores apenas a segunda forma é diretamente aplicável pois não é possível garantir em que posição do vetor estará o primeiro elemento da lista. Para fazer a primeira forma funcionar com vetores seria preciso ter um mecanismo que troque elementos de posição, de modo a ter sempre o início da lista numa certa posição do vetor (a primeira, por exemplo). Isso claramente aumenta o trabalho com inserções ou remoções.

9.1.2 Listas estáticas e listas dinâmicas

No início deste capítulo indicamos que listas são implementadas usando tanto estruturas dinâmicas como estáticas. No capítulo anterior se apresentou um exemplo simples de listas usando estruturas dinâmicas. Falta um exemplo de como se implementa uma lista usando uma estrutura estática, usando vetores.

A diferença entre listas estáticas e dinâmicas está na forma como as estruturas são declaradas. Como exemplo, consideremos os dados constantes no cadastro de frutas apresentado no último capítulo. A arquitetura simplificada desse cadastro usando estruturas dinâmicas é apresentada na figura 9.3. Nela é possível observar que a ligação entre os elementos da lista é feita pelo ponteiro `prox`.

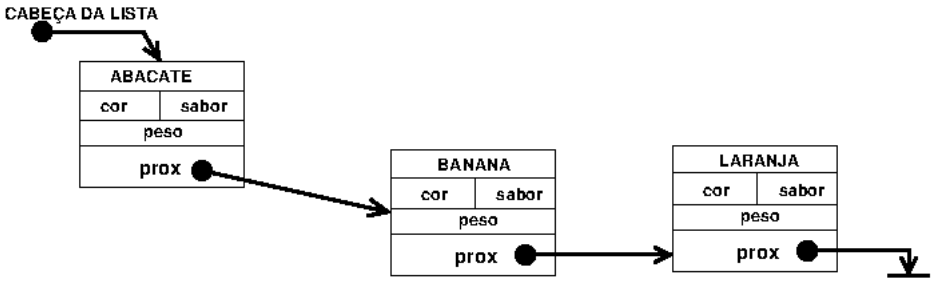

Figura 9.3: Arquitetura dos elementos de lista ligada com estrutura dinâmica.

Já para listas usando estruturas estáticas a ligação entre elementos é feita indicando-se a posição, no vetor, do próximo elemento da lista. Na figura 9.4 se apresenta a lista em um vetor, em que no lugar do campo ponteiro temos um campo ocupado pelo índice da posição que o próximo elemento ocupa no vetor.

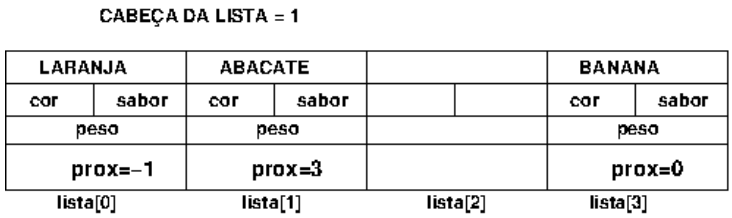

Figura 9.4: Arquitetura dos elementos de lista ligada com estrutura estática.

O processo para inserir ou remover um elemento da lista é, funcionalmente, equivalente ao realizado para estruturas dinâmicas. As diferenças são apenas o uso de índices do vetor em vez de ponteiros e o fato da posição a ser ocupada ou desocupada fazer parte do vetor, tornando alocações e desalocações de memória desnecessárias.

Do ponto de vista da implementação desses índices, a estrutura apresentada na Figura 9.4 usou um campo dela própria para armazenar o índice para a próxima posição da lista. Outra abordagem, **não aconselhável** pois pode criar confusão entre elementos, faz uso de dois vetores, um que armazena a lista e outro que armazena os índices de ligação, sendo que posições de mesmo índice dizem respeito ao mesmo elemento. A declaração de estruturas de lista para essas duas versões é apresentada na Figura 9.5. Para a `struct frutaria` se coloca o índice como parte integrante de cada elemento da lista. Já para a `struct frutaria2,` o que se faz é usar um vetor de índices auxiliar, denominado `vetor_indices`. Nele, o valor encontrado na posição i indica a posição do elemento seguinte ao elemento da posição i do vetor contendo a lista de frutas.

```
// índice na própria estrutura

typedef struct frutaria {
  char nome[20];
  int cor, sabor;
  int prox;
} Fruta;
Fruta cadastro[200];
```

```
// usando vetor de índices

typedef struct frutaria2 {
  char nome[20];
  int cor, sabor;
} Fruta;
Fruta cadastro2[200];
int vetor_indices[200];
```

Figura 9.5: Declaração dos indexadores para lista estática, usando a própria estrutura e usando vetor de índices.

9.1.3 Tipos básicos de listas

A diferenciação entre os vários tipos de lista é feita a partir da forma em que os componentes da lista se relacionam. Teoricamente se pode estabelecer qualquer relação entre elementos armazenados na memória, mas as listas se caracterizam por manter uma relação que possa ser linearizada. Isso vem em contraposição a relações mais complexas, que levam a estruturas planares (árvores binárias) ou espaciais (grafos, árvores k-dimensionais). As listas básicas englobam os seguintes tipos:

1. **Listas ligadas**, que na verdade são constituintes de qualquer outro modelo de lista, e são compostas apenas por elementos ligados um ao outro, sem qualquer restrição adicional, exceto o fato de que "caminhar" entre os elementos da lista só pode ocorrer do início para o final da lista.

2. **Listas ordenadas**, que são um caso especial de listas ligadas, em que se estabelece uma relação de ordem entre elementos consecutivos da lista, isto é, um elemento em uma dada posição possui valor de ordenação maior que seu antecessor na lista e menor que seu sucessor se a ordenação for crescente, ou o contrário disso se for decrescente.

3. **Listas duplamente ligadas**, que são estruturas em que se pode caminhar na lista nos dois sentidos, isto é, do início para o fim ou do fim para o começo, necessitando portanto indicadores para as duas extremidades.

4. **Listas circulares**, que são listas ligadas em que seu início está ligado ao seu final, formando então um círculo. Embora se possa argumentar que isso leva à perda da linearidade, é preciso entender que como os marcadores de início e final de lista são mantidos, então é possível caracterizar uma estrutura linear de relação entre seus elementos a qualquer momento.

5. **Filas**, que são listas ligadas em que se acrescenta a restrição de que a inserção de um novo elemento pode ocorrer apenas em seu final e que um elemento apenas pode ser retirado da lista se estiver em sua primeira posição. Essa restrição faz com que filas sejam chamadas de estruturas FIFO, de *First-In, First-out*.

6. **Pilhas**, que são listas ligadas em que se define que tanto a inserção quanto a remoção de um elemento apenas pode ser feita sobre o final da lista, chamado de topo. Em alguns textos as pilhas são chamadas de LIFO, de *Last-In, First-Out*.

Destas estruturas apenas as listas ordenadas não serão explicitamente examinadas aqui, pois a única diferenciação nelas é a necessidade de ordem. Toda a organização estrutural de listas ordenadas segue exatamente a mesma organização das demais listas apresentadas nas próximas seções.

Da descrição apresentada é possível inferir que, na prática, todas as formas de lista são variações das listas ligadas simples. Assim, aproveitaremos a descrição de listas ligadas para mostrar como listas podem ser implementadas tanto usando estruturas estáticas ou estruturas dinâmicas.

No exame de cada tipo de lista apresentaremos os algoritmos para fazer a inserção e remoção de um elemento na lista. Como listas são estruturas para armazenamento de dados, além da inserção e remoção é importante se definir métodos para busca por determinado elemento na lista. Assim, indicaremos também como a busca pode ser feita em cada lista.

9.2 Listas ligadas simples

Nas próximas páginas serão apresentadas implementações das operações de inserção e remoção de elementos numa lista usando estruturas estáticas. A implementação dessa lista com estruturas dinâmicas foi apresentada no capítulo anterior, não sendo repetida aqui. Reforçamos que o uso de listas estáticas é apresentado aqui apenas para o caso de listas ligadas básicas. Para os demais tipos de listas serão apresentadas apenas as implementações usando estruturas dinâmicas, muito embora possam também ser implementadas com estruturas estáticas, usando índices no lugar de ponteiros.

9.2.1 Inserção em lista simples

A inserção de um novo elemento em uma lista ligada pode ocorrer em qualquer ponto da lista. Como vimos com o código apresentado na Figura 8.2 (pág. 163), a inserção pode ocorrer no começo da lista, ou entre dois elementos dela ou como seu último elemento. A Figura 9.6 apresenta a sequência de ações executadas para a inserção de um elemento no meio de uma lista.

Neste caso, em (a) temos o elemento `DADO C`, que deve ser inserido entre `DADO B` e `DADO D`.

A primeira ação a ser executada é fazer com que `DADO C` passe a apontar para `DADO D`, como aparece em (b), pois assim ele passa a preceder `DADO D`.

Em seguida temos que fazer com que `DADO B` se torne o antecessor de `DADO C`, o que é feito mudando o valor de seu indexador para que aponte para `DADO C`, o que é apresentado em (c), concluindo a inserção do novo elemento.

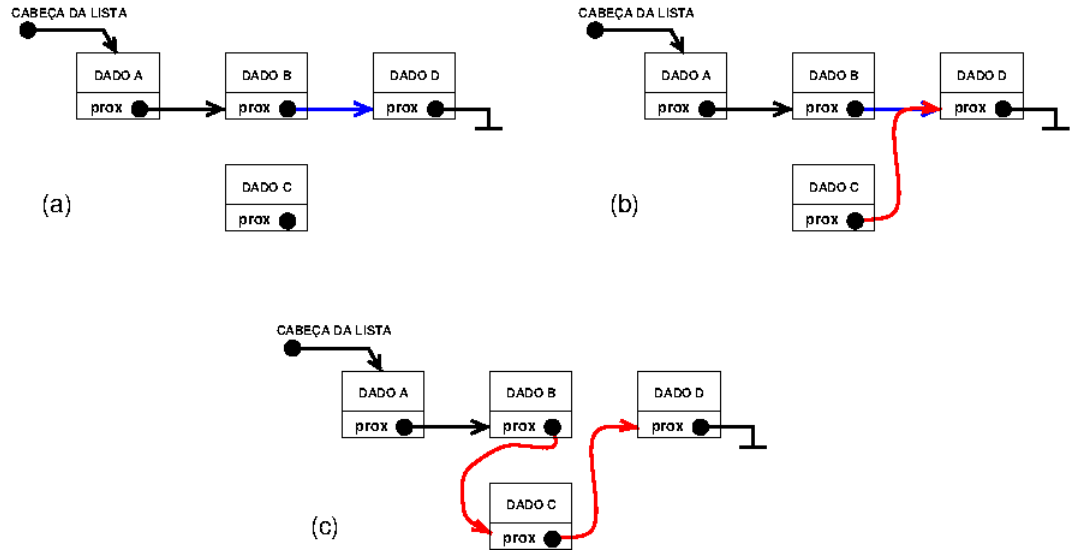

Figura 9.6: Procedimento para inserção de um elemento no meio de uma lista.

Na Figura 9.7 temos o código C para fazer a inserção na versão estática da lista de frutas. Nesse código o primeiro comando `while` percorre o vetor procurando por uma posição vazia, que será igual a *pos*. Em seguida a lista é percorrida pelo segundo comando `while`, até que se encontre a posição correta em que o elemento aparecerá na lista (sua ordem dentro da lista e não no vetor). Uma vez localizada a posição do novo elemento na lista, então se faz o ajuste dos valores do campo `prox` de seus vizinhos na lista e do elemento inserido, cuidando de atualizar o marcador do início se o elemento for inserido como primeiro da lista.

Nessa implementação usamos dois valores constantes para marcar posições desocupadas no vetor e também o final da lista. Primeiro, o final da lista é delimitado por um valor zero no campo `prox`, o que faz com que a posição zero do vetor esteja sempre desocupada, marcando o final da lista. O segundo valor constante é ter um valor negativo para marcar posições vazias no vetor, exceto a posição zero, que ficará sempre desocupada já que o valor zero marca o final da lista. Apesar de desperdiçar uma posição no vetor, essa solução é bem mais prática pois fica fácil identificar posições vazias e também o final da lista.

Para listas dinâmicas, a inserção parte do princípio de que a posição a ser ocupada pelo novo elemento é uma nova posição na memória. Isso significa que o primeiro `while` é substituído pela alocação do espaço usando `malloc` ou pelo recebimento dessa posição, caso ela tenha sido alocada externamente à função. O segundo comando `while` segue o mesmo procedimento, porém considerando que para estruturas dinâmicas se deve seguir o ponteiro que liga os elementos da lista. Nesse caso o limite no processo de busca, evitando erros de execução, é

```
1  void insereFruta(Fruta cad[200], Fruta nova, int *head)
2  // head aponta para um inteiro contendo o índice do começo da lista
3   {int i, pos, antes;
4
5     pos = 1; // posição 0 marca o final da lista
6     while (pos<200 && cad[pos].prox >= 0)
7         pos++;    // procura posição vazia no vetor
8     strcpy(cad[pos].nome,nova.nome);
9
10 // inserir aqui código para copiar "nova" na posição vazia
11    // do vetor, sendo usado cad[pos] a partir disso
12    i = *head;
13    while(i > 0 && strcmp(cad[i].nome, cad[pos].nome) < 0)
14    { antes = i;
15      i = cad[i].prox;
16    }
17    cad[pos].prox = i;
18    if (i == *head) // vai inserir no começo da lista
19        *head = pos;   // posição pos será novo começo da lista
20      else
21        cad[antes].prox = pos;
22  }
```

Figura 9.7: Inserção de um elemento em lista estática, usando o tipo de dados estruturado "Fruta" apresentado na página 178.

dado pela identificação de um ponteiro para `NULL`. Por fim, as ligações do novo elemento são feitas com o acerto de ponteiros, como vimos no capítulo anterior (Figura 8.2, na página 163).

9.2.2 Remoção em lista simples

Para a remoção de um elemento da lista se deve cuidar para não perder a continuidade da mesma. Isso resulta na sequência de ações apresentada na Figura 9.8, em que o elemento `DADO C` deve ser removido. O primeiro passo é fazer com que `DADO B` passe a apontar para `DADO D`, como aparece em (b). Depois basta remover de fato `DADO C`, o que é apresentado em (c).

Para listas estáticas essas operações são aplicadas sobre os campos de indexação da lista. Assim, em vez de remover uma posição do vetor o que se faz é marca-la como vazia, o que pode ser feito tanto limpando os conteúdos dela como colocando um valor pré-determinado no campo que indica seu sucessor.

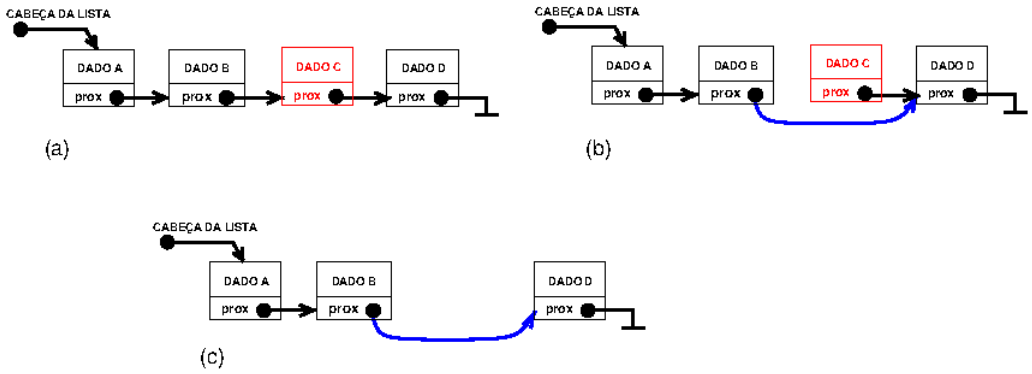

Figura 9.8: Procedimento para remoção de um elemento no meio de uma lista.

A Figura 9.9 contém o código para remoção em uma lista estática. No laço *while* se busca a posição que contém o elemento a ser removido. A saída do laço ocorre em três hipóteses: não se encontrou o elemento (`i=0`) ou foi encontrado e é o primeiro da lista ou foi encontrado em uma outra posição qualquer dela. Essas condições são examinadas sucessivamente nos comandos *if* seguintes.

Para o caso de uma implementação usando estruturas dinâmicas as modificações são simples. O código apresentado na Figura 9.10 mostra como a remoção é feita neste caso. No laço *while* a diferença está em usarmos ponteiros para caminhar na lista, implicando na verificação do final da lista com um ponteiro nulo. Após a saída do laço de busca, no comando *if* se faz o acerto dos ponteiros seguindo o procedimento descrito na Figura 9.8. A principal diferença está na "remoção" efetiva do elemento, que no caso de estruturas dinâmicas implica no uso do comando *free* para efetivamente liberar o espaço ocupado na memória.

9.2.3 Busca em listas simples

A busca em uma lista ligada é bastante simples, independente de ser implementada com estrutura estática ou dinâmica. Uma diferenciação deve ser feita caso a lista seja ordenada, quando se estabelece um critério de parada adicional, como será visto a seguir. Fora isso o processo de busca parte do elemento inicial da lista e, para cada elemento da lista, verifica se é o elemento procurado ou não.

No caso do elemento verificado ser o procurado, a função de busca tipicamente retorna a posição localizada. Em caso negativo, a busca continua passando-se para o próximo elemento da lista, até chegar-se ao final dela.

O código da função de busca usando lista estática aparece na Figura 9.11, destacando-se o laço `while` em que se faz a busca. Após o laço se acrescentou um comando `if` para mostrar mais claramente quando se encontrou o elemento

```
1   void removeFruta(Fruta cad[200], char *fruta, int *head)
2  // head aponta para um inteiro contendo o índice do começo da lista
3   {int i, antes;
4
5     i = *head;
6     while(i >= 0 && strcmp(cad[i].nome, fruta) < 0) // Busca a fruta
7     { antes = i;
8       i = cad[i].prox;
9     }
10    if (strcmp(cad[i].nome, fruta)) return; // não encontrou
11    if (i == *head) // vai remover a primeira fruta da lista
12        *head = cad[i].prox;  // registra novo começo da lista
13      else  // removendo elemento no meio da lista
14        cad[antes].prox = cad[i].prox;  // acerta novo próximo
15    cad[i].prox = -1;  // marca posição "removida" como livre
16    return;
17  }
```

Figura 9.9: Remoção de um elemento em lista estática.

```
1   void removeFruta(Fruta *head, char *fruta)
2   // head é o ponteiro para primeiro elemento da lista
3   {Fruta *aux, *antes;
4
5     aux = head;
6     while(aux && strcmp(aux->nome, fruta)) // Busca pela fruta
7     { antes = aux;
8       aux = aux->prox;
9     }
10    if (aux == head) // vai remover a primeira fruta da lista
11        *head = aux->prox;  // registra novo começo da lista
12      else  // removendo elemento no meio da lista
13        antes->prox = aux->prox;  // acerta novo próximo
14    free(aux);  // libera posição ocupada pelo elemento removido
15  }
```

Figura 9.10: Remoção de um elemento em lista dinâmica.

buscado (`aux` maior que zero) ou quando não se encontrou, retornando -1.

Para listas dinâmicas o processo é semelhante, bastando mudar o critério de parada do laço *while* para `while(aux && strcmp(aux->nome,fruta))` e a condição do comando *if*, caso seja usado, para `if(aux)`. Isso, óbvio, além das

```
1   int buscaFruta(Fruta cad[200], char *fruta, int head)
2      // head é o indicador para primeiro elemento da lista
3   {int aux;
4
5     aux = head;
6     // Procura pela fruta a ser removida
7     while (aux > 0 && strcmp(cad[aux].nome, fruta))
8           aux = cad[aux].prox;  // Avança para próximo da lista
9     if (aux > 0)
10         return (aux);  // retorna a posição encontrada
11      else
12         return (-1);  // retorna 0 para indicar que não encontrou
13  }
14  // Na prática este if é desnecessário. Ele foi colocado apenas
15  // para mostrar quando se encontrou de fato o elemento buscado.
16  // Uma implementação mais simples teria apenas "return(aux);"
```

Figura 9.11: Busca por um elemento em lista estática.

modificações ligadas ao fato de se usar um ponteiro e não um índice do vetor.

9.2.4 Busca em listas ordenadas

Quando uma lista está ordenada é possível interromper a busca antecipadamente, mesmo quando o elemento procurado não faz parte da lista. Isso é possível pois caso a lista esteja ordenada crescentemente, por exemplo, e a busca seja por um determinado valor M_i, ao se chegar a um elemento de valor M_j, sendo $M_j > M_i$, se torna desnecessário continuar a busca. Isso é óbvio pois todos os elementos na parte ainda não examinada da lista também terão valores maiores que M_i.

Assim, o teste executado no comando *while* deve ser modificado para verificar se o elemento buscado ainda antecede o elemento atual da lista. Aqui "anteceder" deve ser avaliado de acordo com o tipo de valor sendo comparado (numérico ou alfabético) e de acordo com a forma de ordenação da lista (crescente ou decrescente). No caso do código apresentado na Figura 9.11, a condição do laço passaria a ser `(aux && strcmp(cad[aux].nome,fruta) < 0)`, e a condição do `if` seria modificada para verificar se `cad[aux]` realmente contém o elemento procurado.

EXERCÍCIOS

1. Considerando uma lista estática ordenada de números inteiros, em que cada

posição armazena o número e o índice do próximo número na lista, preencha o conteúdo do vetor com a inserção da seguinte sequência:

54, 16, 77, 98, 34, 8, 96, 80, 45, 23, 6

2. Como fica o vetor da questão anterior se os números 77, 45 e 34 forem removidos da lista e for inserido o número 2?

3. Modifique os códigos de inserção e remoção apresentados no texto para que se use um índice negativo como indicador de final da lista e posições desocupadas sejam identificadas pelo valor do dado armazenado (posição 0 do vetor deve ser usada).

9.3 Listas duplamente ligadas

Elementos de listas duplamente ligadas possuem dois indicadores para os elementos vizinhos. Um indicador aponta para seu antecessor e o outro aponta para seu sucessor na lista. O objetivo de se usar ligação dupla é permitir que se percorra a lista em qualquer sentido, ou seja, do começo para o fim ou do fim para o começo. Essa possibilidade é bastante interessante em listas ordenadas com muitos elementos, que agora podem ser percorridas a partir da extremidade "mais próxima" do elemento a ser inserido, removido ou buscado.

NOTA - Na descrição dos métodos de operação de uma lista duplamente ligada consideraremos apenas a sua versão dinâmica, com ponteiros. Deve ficar claro, entretanto, que elas podem ser também facilmente implementadas usando estruturas estáticas, porém com dois índices, um para o elemento anterior e outro para o elemento posterior.

9.3.1 Inserção de um elemento em lista duplamente ligada

Para a inserção de um elemento em uma lista duplamente ligada é preciso acertar vários ponteiros de forma a garantir o correto duplo apontamento. A Figura 9.12(a) mostra a estrutura da lista em que se quer inserir um elemento (`DADO C`) entre os elementos `DADO B` e `DADO D`. Nela as setas azuis representam os ponteiros que terão que ser alterados. O primeiro passo, visto em 9.12(b), consiste em copiar esses ponteiros para os respectivos campos de `DADO C`, como indicado pelas setas em vermelho. A finalização da inserção é feita então copiando-se o endereço de `DADO C` para o campo `prox` de `DADO B` e para o campo `prev` de `DADO D`, como visto na Figura 9.12(c).

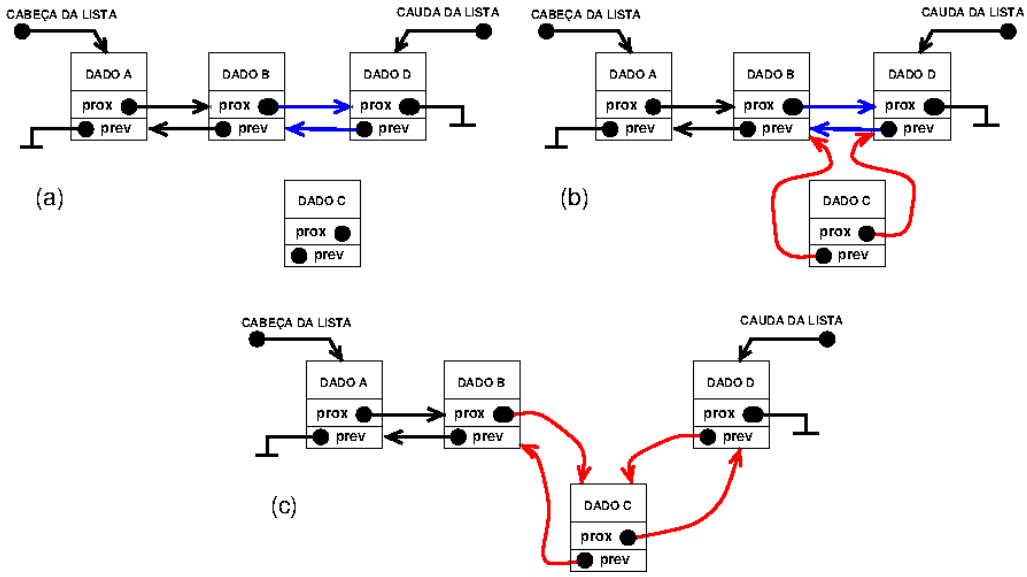

Figura 9.12: Procedimento para inserção de um elemento no meio de uma lista duplamente ligada.

Diferente das implementações apresentadas na seção 9.2, aqui não usaremos uma estrutura de dados específica, o que permite generalizar as funções de manipulação da lista[1]. Embora pareça menos "prático", os códigos apresentados podem ser facilmente modificados para casos específicos que usem tais listas.

Na Figura 9.13 temos o código genérico para a inserção de um elemento em uma lista duplamente ligada ordenada. Nele se considera como parâmetros de entrada os ponteiros para início e final da lista, assim como um ponteiro para a estrutura em que se armazenou o elemento a ser inserido. Nessa função são utilizadas duas funções abstratas para auxiliar a manipulação da lista. A primeira função, `maisProximo`, verifica se o elemento a ser inserido está mais próximo do começo da lista (retornando valor diferente de zero) ou do final da lista. Já a função `naoChegou` verifica se já localizou o ponto correto de inserção.

A função de inserção, propriamente dita, começa verificando se a lista está vazia, inserindo o elemento como o começo e final da lista caso isso seja verdade (primeiro *if*).

O segundo comando *if* verifica se o elemento a ser inserido está mais próximo do começo ou do final da lista. Deve-se observar que isso nem sempre é necessário, pois dependendo da aplicação a inserção pode ocorrer em uma extremidade específica. A escolha em fazer essa diferenciação é para generalizar nossa solução.

[1]Vale observar que essa mesma forma de construção será adotada para os demais tipos de lista neste capítulo.

```
1  void insereDupla(Lista **head, Lista **tail, Lista *elem)
2  // elem é o endereço do elemento a ser inserido na lista
3  {Lista *aux;
4
5    if (!(*head)) // Lista está vazia
6        { *head = *tail = elem;       return;  }
7    if (maisProximo(elem,*head,*tail))  // define o ponto de partida
8        { aux = *head;
9          while (naoChegou(elem, aux, 1)) // procura local p/ inserção
10             aux = aux->prox;
11         elem->prox = aux;  elem->prev = aux->prev;  // Fig 9.10(b)
12         aux->prev = elem;  // Figura 9.10(c)
13         if (aux == *head) // se inserir antes do começo
14             *head = elem; // muda o começo da lista
15           else
16             (elem->prev)->prox = elem;
17       }
18      else
19        { // Repete o processo, trocando head por tail e todas
20          // referências aos ponteiros prox e prev entre si
21        }
22    return;
23  }
```

Figura 9.13: Inserção de um elemento em lista duplamente ligada.

No caso da inserção estar mais próxima do começo da lista o primeiro passo é apontar para *head* e percorrer a lista até encontrar o ponto de inserção (função `naoChegou`). Uma vez determinado o ponto de inserção se passa a executar as ações previstas na Figura 9.12 para acerto dos ponteiros, fazendo um teste adicional para verificar se a inserção irá alterar o ponteiro para o início da lista.

9.3.2 Remoção de elemento em lista duplamente ligada

Assim como o procedimento de inserção não difere muito da inserção em uma lista ligada simples, a remoção também é semelhante ao que já fizemos. A diferença, como esperado, é que agora são dois ponteiros que devem ser redirecionados, como mostrado na Figura 9.14.

No exemplo da figura se deve retirar o elemento `DADO C`. Para fazer a remoção é preciso primeiro refazer o direcionamento dos elementos que apontam para ele. Isso é mostrado em 9.14(b), com as setas azuis representando os novos valores

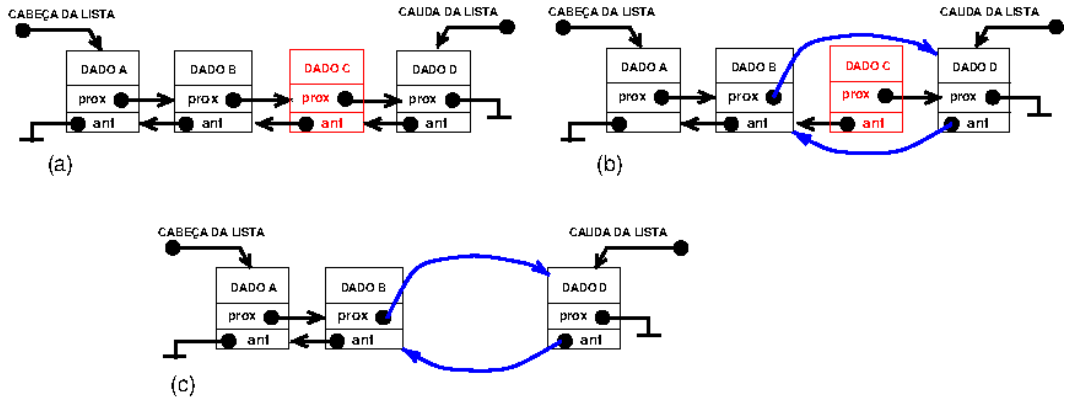

Figura 9.14: Procedimento para remoção de um elemento no meio de uma lista duplamente ligada.

de `prox` em seu antecessor (`DADO B`) de `ant` em seu sucessor (`DADO C`). O passo seguinte, visto em 9.14(c), remove efetivamente `DADO C`, deixando a lista na sua configuração final, em que agora `DADO B` passa a ser o antecessor de `DADO D`.

A implementação desses passos é vista na Figura 9.15, em que se considera uma lista com elementos de um tipo abstrato genérico. Antes de remover o elemento é necessário encontrá-lo na lista, antes de qualquer outra operação, primeiro verificando se temos uma lista vazia ou contendo apenas um elemento.

Os casos especiais aqui são a lista vazia, quando deve retornar sem fazer nada, ou quando elemento a ser retirado for o primeiro ou último da lista, quando deverá modificar o marcador respectivo de início ou final da lista. Um caso especial desse tipo ocorre quando a lista tem apenas um elemento, que ao ser removido torna a lista vazia, implicando na modificação dos dois marcadores.

A primeira parte do código mostra os testes sobre a lista vazia (*if (! *head)*). Na sequência se determina a partir de que extremo se busca o elemento a ser retirado, com a chamada da função *maisProximo*. Deve ser observado que isso é útil para listas ordenadas muitos grandes, quando a diferença de tempo para caminhar a partir de um lado ou de outro pode ser significativa.

Uma vez definido o ponto de partida para buscar o elemento a ser removido, então se caminha na lista até encontrar o elemento, ou chegar até a condição de parada prevista caso o elemento não esteja na lista. Essa condição pode ser atingir o outro extremo da lista ou passar da posição que conteria o elemento no caso da lista ser ordenada, como foi considerado aqui. Tendo encontrado o elemento se faz a sua remoção de fato, nas linhas 11 até 20, tomando o cuidado de possivelmente atualizar os ponteiros para a cabeça e final da lista, se necessário.

```
1  void removeDupla(Lista **head, Lista **tail, Lista *elem)
2  // elem contém dados do elemento a ser removido
3  {Lista *aux, *antes;
4
5    if (!(*head)) { puts("Lista vazia"); return;} // Lista está vazia
6    if (maisProximo(elem,*head,*tail))  // define o ponto de partida
7        { aux = *head;                  // partindo do início da lista
8          while (naoChegou(elem, aux, 1))  // procura pelo elemento
9              aux = aux->prox;
10         if (elem->key == aux->key) // se encontrou remove
11             { if (aux == *head)
12                   *head = aux->prox;  // atualiza começo se preciso
13               else
14                 (aux->prev)->prox = aux->prox;
15             if (aux == *tail)
16                 *tail = NULL;    // limpa a lista se ficar vazia
17               else
18                 (aux->prox)->prev = aux->prev;
19             free (aux);
20             }
21           else printf("%s nao encontrado\n",elem->key);
22       }
23     else
24       { // Repete o processo, trocando head por tail e todas
25         // referências aos ponteiros prox e prev entre si
26       }
27 }
```

Figura 9.15: Remoção de um elemento em lista duplamente ligada.

9.3.3 Busca em lista duplamente ligada

A busca numa lista duplamente ligada depende da organização dos dados da lista. Se ela estiver ordenada a busca pode, ou deve, fazer uso dessa informação, partindo da extremidade possivelmente mais próxima do elemento procurado.

Caso a lista não esteja ordenada o processo é mais simples, partindo do começo da lista e caminhando nela até que se encontre o elemento ou se chegue ao seu marcador de final (cauda da lista). Para listas ordenadas a busca deve considerar a maior proximidade de um dos extremos, reduzindo o espaço de busca se o elemento não for encontrado, como se fez nas funções para inserção e remoção. Seu código não será apresentado aqui pois é equivalente ao processo de busca pelo elemento a ser removido da função para remoção, apresentada na Figura 9.15.

9.4 Listas circulares

Listas circulares são uma modificação significativa nas listas vistas até agora pois nelas o começo da lista é ligado ao seu final. Mesmo com essa ligação as listas circulares podem ter tanto ligação simples como dupla, dependendo do tipo de aplicação a que se destinam. Também podem ser ordenadas ou não. Aplicações típicas de listas circulares envolvem situações em que seus elementos são "visitados" múltiplas vezes e de modo sequencial. Uma situação desse tipo é o atendimento em *round-robin*, em que se organiza os elementos em forma circular, que são atendidos na sequência em que aparecem no círculo, reiniciando o atendimento cada vez que se completa uma volta.

9.4.1 Inserção de um elemento em lista circular

Como para qualquer lista, a inserção em lista circular começa identificando o ponto em que ocorrerá a inserção. Aqui é importante indicar que listas circulares podem ser tratadas de dois modos. Primeiro, como uma lista sequencial, com um marcador de começo da lista que será sempre usado para seu acesso. Segundo, com um marcador para o último elemento inserido e outro para o último elemento "lido", o que é tratado como um *buffer* circular. Na Figura 9.16 se ilustra as etapas envolvidas para uma lista do primeiro tipo, com ligação simples. A variável `cabeca` aponta para o elemento atualmente no início da lista e que o novo elemento será inserido em um ponto conveniente.

Como é possível ver, primeiro se atualiza o ponteiro do elemento a ser inserido para que aponte para o elemento que o antecederá (Figura 9.16(b)). Depois se faz esse elemento apontar para o elemento a ser inserido, resultando na lista mostrada na Figura 9.16(c). Se a lista fosse duplamente ligada, as modificações necessárias dizem respeito ao duplo apontamento, como ocorre com listas duplamente ligadas. O código apresentado na Figura 9.17 ilustra as operações indicadas na figura anterior. Nele a função `naoChegou` verifica se a posição de inserção foi encontrada, retornando o valor 0 quando isso ocorrer.

9.4.2 Remoção de um elemento em lista circular simples

A remoção em uma lista circular segue o mesmo procedimento adotado para as listas examinadas até agora. Apresentamos aqui a remoção de um elemento considerando uma lista circular com ligação simples. O processo é simples, bastando primeiro localizar o elemento da lista que deve ser removido, como aparece destacado na Figura 9.18(a). A partir disso é necessário fazer com que o elemento anterior na lista passe a apontar para o elemento apontado por aquele que será retirado, como visto na Figura 9.18(b). Finalmente o elemento pode ser removido

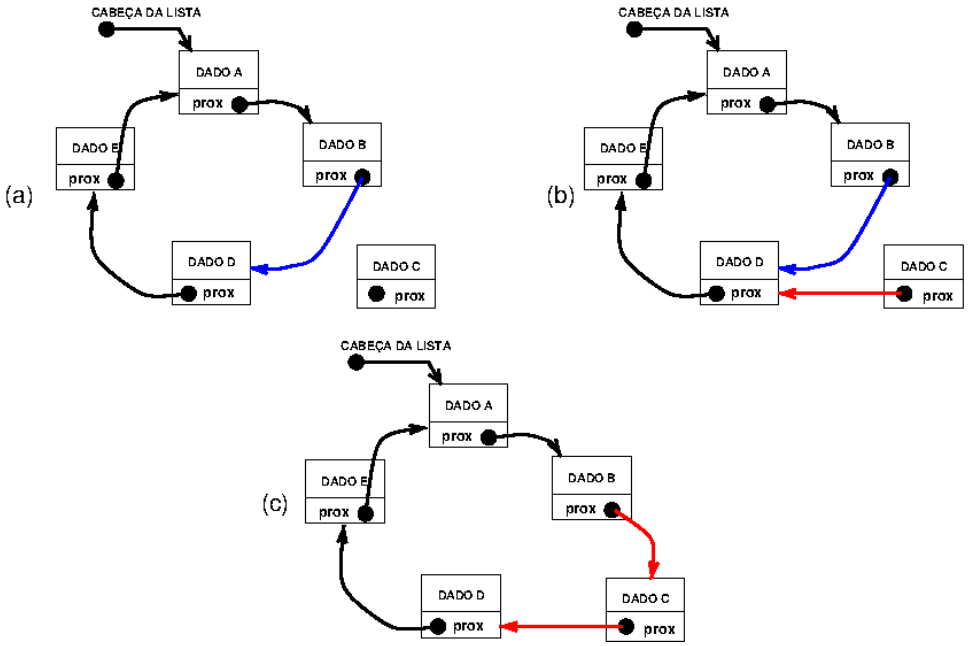

Figura 9.16: Procedimento para inserção de um elemento em uma lista circular.

da lista, ou repassado a alguma outra funcionalidade do programa.

O procedimento de remoção deve cuidar, adicionalmente, para que o processo de busca pelo elemento a ser removido termine mesmo que ele não seja encontrado. O código apresentado na Figura 9.19 considera que a remoção é o objetivo final da operação. Quando temos *buffers* circulares o elemento removido é retornado como resposta da função de remoção. Os parâmetros passados são ponteiros para o elemento inicial da lista e para uma estrutura que contém dados sobre o elemento a ser retirado.

O código começa com dois testes para casos limites, que são a lista vazia e do elemento a ser removido ser o único da lista. No primeiro caso apenas se dá o retorno indicando que o elemento não foi encontrado. No segundo caso é preciso atualizar o ponteiro para início da lista, além de efetivamente remover o elemento.

Após tratar os casos limites se passa ao tratamento da remoção de um elemento qualquer da lista. Para isso se usa os ponteiros `atual` e `antes` para caminhar na lista. O processo de busca continua até que se encontre o elemento (função `naoChegou` retornando zero) ou que o ponteiro `atual` volte a apontar para o começo da lista.

```
1  void insereCirc(Lista **cabeca, Lista *elem)
2  // elem aponta para os dados do elemento buscado e elem->prox=elem
3  {Lista *atual = (*cabeca), *antes = (*cabeca);
4
5    if (!(*cabeca)) // Lista está vazia
6        { (*cabeca) = elem;          return;  }
7    if (atual->prox == atual)  // apenas um elemento
8        { elem->prox =  atual;
9          atual->prox = elem;
10          if (elem->key < (*cabeca)->key)
11              (*cabeca) = elem;
12          return;
13       }
14   // Procura posição para inserção
15   while (antes->prox != (*cabeca) && naoChegou(elem,atual,1))
16   { antes = atual;    atual = atual->prox;  }
17   if (atual == (*cabeca) && elem->key < atual->key)
18       { antes = atual->prox;
19         while (antes->prox != atual)
20             antes = antes->prox;
21         antes->prox = elem;
22         elem->prox = (*cabeca);
23         (*cabeca) = elem;
24         return;
25       }
26   elem->prox = atual;  // Figura 9.14(b)
27   antes->prox = elem;  // Figura 9.14(c)
28   if (elem->key < (*cabeca)->key))
29       (*cabeca) = elem;
30   return;
31 }
```

Figura 9.17: Inserção de um elemento em lista circular com ligação simples.

9.4.3 Busca em lista circular

A busca em uma lista circular é semelhante ao procedimento adotado para listas simples. A diferença aqui é identificar com precisão o critério de parada, uma vez que não teremos um "final" da lista propriamente dito caso o elemento procurado não esteja na lista. Assim, o critério a ser adotado quando não se localiza o elemento é verificar que se voltou ao ponto de partida da busca.

Com isso, o procedimento de busca deve começar marcando o primeiro ele-

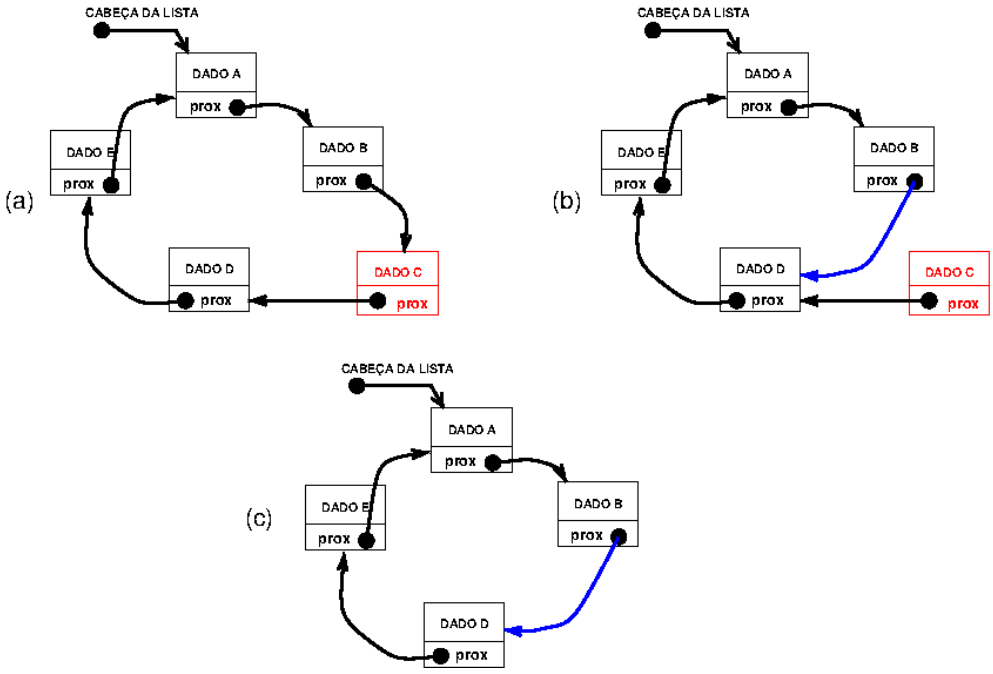

Figura 9.18: Procedimento para remoção de um elemento em uma lista circular.

mento visitado. Tipicamente esse elemento é apontado pela variável que armazena a posição atual de inserção na lista. A partir desse elemento a busca continua até que se encontre o elemento ou que se retorne ao elemento marcado. Esse procedimento está apresentado na Figura 9.20, sendo que o laço da linha 10 é o responsável pelo processo de busca. O teste da linha 12 define, ao final do processo, se o elemento foi encontrado ou não.

9.4.4 Tratamento de *buffers* circulares

A implementação de *buffers* circulares é, comparativamente, mais simples. Isso ocorre pela existência de dois marcadores, sendo um para os elementos a serem inseridos e outro para os que serão "lidos". O único cuidado a ser tomado é evitar que o marcador de inserção ultrapasse o marcador de leitura, pois nesse caso passaríamos a inserir novos dados sobre dados já inseridos mas que ainda não foram utilizados.

A inserção em um *buffer* é facilitada se o marcador de inserção apontar para o último elemento inserido. Nesse caso podemos adotar duas posturas para a inserção:

i Sobrescrever elementos já lidos se o marcador de leitura estiver mais adiante;

```
1  void removeCirc(Lista **cabeca, Lista *elem)
2   // elem aponta para os dados do elemento a ser removido
3  {Lista *atual=(*cabeca), *antes=(*cabeca);
4
5    if (! *cabeca) // Lista está vazia
6        { puts("lista vazia"); return; }
7    // retira único elemento da lista
8    if (atual->prox == atual && elem->key == atual->key)
9        { (*cabeca) = NULL;
10         free(atual);
11         return;
12       }
13   // Procura elemento a ser removido nos demais casos
14   while (atual->prox != (*cabeca) && naoChegou(elem,atual,1))
15   { antes = atual;         atual = atual->prox;   }
16   // Verifica se de fato encontrou o elemento a ser removido
17   if (elem->key == atual->key)
18       { if (atual != (*cabeca) )
19             antes->prox = atual->prox;  // Figura 9.16(b)
20           else
21             { while (antes->prox != (*cabeca))
22                   antes = antes->prox;
23               antes->prox = atual->prox;
24               (*cabeca) = atual->prox;
25             }
26         free(atual);  // Figura 9.16(c)
27       }
28     else
29       puts("Elemento não encontrado");
30   return;
31 }
```

Figura 9.19: Remoção de um elemento em lista circular com ligação simples.

ii Alocar um novo espaço para inserção se o próximo elemento estiver apontado pelo marcador de leitura.

Já o processo de leitura também é simples, bastando deixar o marcador apontando para o próximo elemento a ser lido. Aqui o cuidado é parar o processo caso se alcance o marcador de inserção de novos elementos no *buffer*, evitando que se acesse novamente elementos já lidos.

```
1   Lista *buscaCircular(Lista *cabeca, Lista elem)
2   // elem aponta para os dados do elemento buscado
3   {Lista *no=cabeca->prox;
4
5     // Verifica se o primeiro da lista é o elemento buscado
6     if (elem->key == cabeca->key)
7         return(cabeca);
8
9     // Se não for continua a busca pelo próximo elemento da lista
10     while (no != cabeca && elem->key < no->key)
11       no = no->prox;
12     if (elem->key == no->key)
13         return(no);   // Elemento foi encontrado
14       else
15         return(NULL); // Não se encontrou o elemento
16  }
```

Figura 9.20: Busca por um elemento em lista circular com ligação simples.

9.5 Filas

Todos os tipos de listas examinados até agora não definem qual a posição de inserção ou retirada de um elemento. Isso significa que um elemento pode ser inserido em qualquer ponto dela, desde que atenda a alguma qualificação para ser colocado naquele ponto. Entretanto, existem aplicações em que as listas apresentam determinado comportamento quanto aos pontos de inserção e remoção, como é o caso de uma fila.

Uma lista do tipo fila é exatamente a representação computacional de uma fila da vida real. Isso significa que um elemento a ser inserido na fila será sempre colocado em seu final. Do mesmo modo, um dado elemento pode ser removido de uma fila apenas se ele for o primeiro elemento dela. Dessas restrições é que surge o nome FIFO (*First-In, First-Out*) para estruturas de fila. Assim, os procedimentos para inserção e remoção são mais simples do que aqueles que vimos até o momento, uma vez que não é necessário "procurar" a posição correta para a operação.

9.5.1 Inserção de um elemento em uma fila

Como indicado, a inserção de um elemento em uma fila ocorre apenas no seu final. Assim, o procedimento adotado é ter um marcador do final da fila, que é passado como parâmetro da função de inserção. Na função de inserção se faz o

elemento do final apontar para o elemento a ser inserido e, na sequência, muda-se o marcador de final para esse novo elemento, como visto na Figura 9.21.

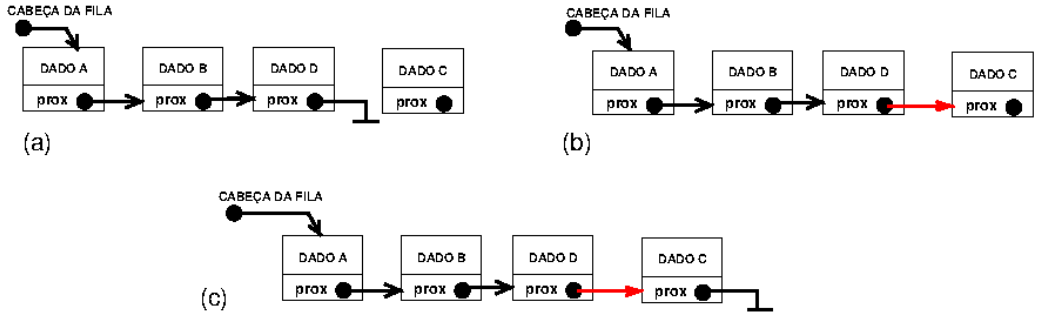

Figura 9.21: Procedimento para inserção de um elemento em uma fila.

Na Figura 9.22 é apresentado o código para a inserção em uma fila genérica. Nessa função se considera que existem dois ponteiros para a fila, um marcando seu início e outro o seu final. Com isso se evita ter que percorrer a fila toda para cada inserção. O único cuidado a ser tomado é quando a fila está vazia, quando se tem que definir também o ponteiro para seu início.

```
1   void InsereElem(Fila **cabeca, Fila **cauda, Fila *elem)
2   // elem aponta para os dados do elemento buscado
3   {Fila *atual, *antes;
4
5     if (! *cauda) // Fila está vazia
6         (*cabeca)->prox = elem;
7     (*cauda)->prox = elem;  // Figura 9.19(b)
8     elem->prox = NULL;      // Figura 9.19(c)
9     return;
10   }
```

Figura 9.22: Inserção de um elemento em uma fila.

9.5.2 Remoção de um elemento em uma fila

Por definição a remoção é sempre feita sobre o primeiro elemento da fila. Como filas são TADs criadas com o propósito de implementar uma fila de atendimento, a remoção do primeiro elemento da fila deve estar associado ao seu uso em alguma outra operação. Assim, as ações necessárias implicam em transferir o primeiro elemento como valor de retorno da função, atualizando o marcador do começo da

fila para apontar o elemento seguinte ao removido.

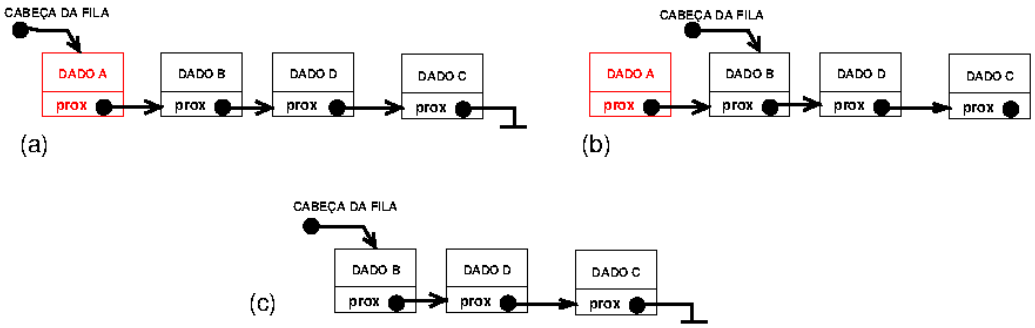

Figura 9.23: Procedimento para remoção de um elemento em uma fila.

O código da Figura 9.24 apresenta o código genérico para a remoção em uma fila. Ele é bastante simples, sendo que o cuidado a ser tomado é quando se retira o último elemento da fila, que se tornará vazia portanto.

```
1   Fila *RemoveElem(Fila **cabeca, Fila **cauda)
2   // elem aponta para os dados do elemento buscado
3   {Fila *elem;
4
5     if (! *cabeca) // Fila está vazia
6         return (NULL);
7     elem = (*cabeca)->prox;
8     (*cabeca)->prox = elem->prox;
9     if (elem->prox == NULL)
10         (*cauda)->prox = NULL;
11    return (elem);
12  }
```

Figura 9.24: Remoção de um elemento em uma fila.

9.5.3 Métodos de Busca

Considerando-se que uma fila é definida pela restrição aos pontos em que elementos são inseridos ou removidos, a busca por um determinado elemento não faz muito sentido. A busca não faz sentido pois **pela definição de fila** um elemento no meio dela não pode ser modificado. Então, buscar por um determinado elemento em uma fila não tem aplicação direta.

Apesar disso, existem variações de filas em que é útil identificar elementos no

meio dela. Esse é o caso, por exemplo, de filas de prioridade (imaginem um banco atendendo uma fila única, mas dando preferência a idosos, por exemplo). Para aplicações desse tipo o que se faz normalmente é modificar o método de inserção, que passa a considerar a prioridade do elemento a ser inserido.

De qualquer modo, fazer a busca em uma fila segue exatamente o mesmo procedimento de busca em uma lista ligada simples. Apenas é preciso lembrar que não se pode modificar o elemento que foi buscado.

9.6 Pilhas

Pilhas representam um outro tipo de lista em que as operações de inserção e remoção devem ser feitas em pontos específicos da lista e, portanto, não se admite operações em elementos que não estejam nessas posições. No caso de pilhas o que se tem é a operação equivalente ao acesso a uma pilha de objetos, em que para se evitar complicações sempre se manipula o objeto no topo da pilha.

Assim, uma pilha representa uma estrutura de dados em que elementos podem ser empilhados (inserção) ou desempilhados (remoção) apenas na extremidade definida como **topo da pilha**. Assim como ocorre com a estrutura de fila, tais restrições levam a simplificações nos métodos para inserção e remoção, como veremos a seguir.

> **NOTA -** Vale observar que essas operações serão denominadas "empilha" e "desempilha", de modo a associar os nomes com as funcionalidades pretendidas. A notação usada em inglês, *push* para empilhar e *pop* para desempilhar, é bastante usada, mesmo em textos em português.

9.6.1 Inserção de um elemento em uma pilha

Como indicado, a inserção de um elemento em uma pilha ocorre apenas em seu topo. Assim, o procedimento adotado é ter um marcador do topo da pilha, que é passado como parâmetro da função de inserção. Como aqui a remoção também ocorre no topo da pilha, é mais prático fazer o elemento a ser inserido apontar para o topo atual e, na sequência, muda-se o marcador de topo para esse novo elemento, como visto na Figura 9.25.

O procedimento para a operação de empilhamento é, portanto, bastante simples. Uma versão genérica do código está apresentada na Figura 9.26. Nele se usou um ponteiro duplo para modificar o endereço apontado pelo topo da pilha, evitando que a função retorne um endereço. Obviamente, isso pode ser modi-

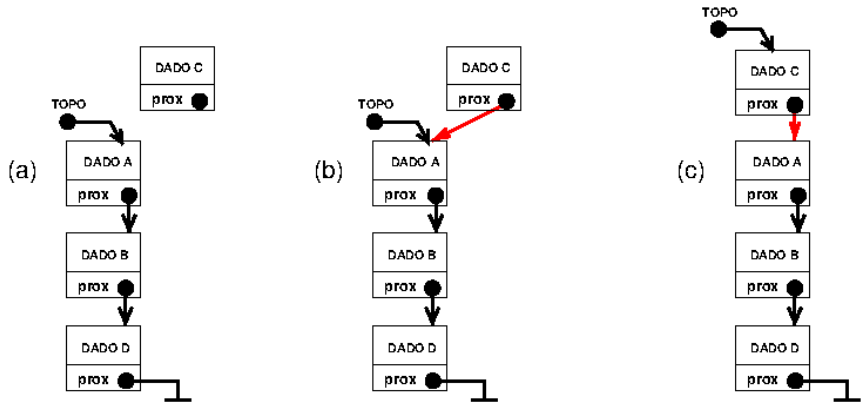

Figura 9.25: Procedimento para inserção de um elemento em uma pilha.

ficado, fazendo com que o ponteiro para o topo seja atualizado como valor de retorno desta função.

```
1   void Empilha(Pilha **topo, Pilha *elem)
2   // elem aponta para os dados do elemento a ser inserido
3   {Fila *atual, *antes;
4
5     if (! (*topo)->prox) // Pilha está vazia
6         elem->prox = NULL;
7       else
8         elem->prox = (*topo)->prox;   // Figura 9.23(b)
9     (*topo)->prox = elem;             // Figura 9.23(c)
10    return;
11  }
```

Figura 9.26: Inserção de um elemento em uma pilha.

9.6.2 Remoção de um elemento em uma pilha

Por definição a remoção é sempre feita sobre o primeiro elemento da fila. Assim, as operações necessárias são transferir o primeiro elemento como valor de retorno da função, atualizando o marcador do começo da fila para apontar o elemento seguinte ao removido.

Na Figura 9.28 se apresenta o código genérico para a remoção em uma fila. Assim como ocorre com filas, o uso de pilhas está normalmente associado a que se use o elemento desempilhado para alguma outra ação. Deste modo, aqui também o desempilhamento implica em que se retorne o elemento como resultado da

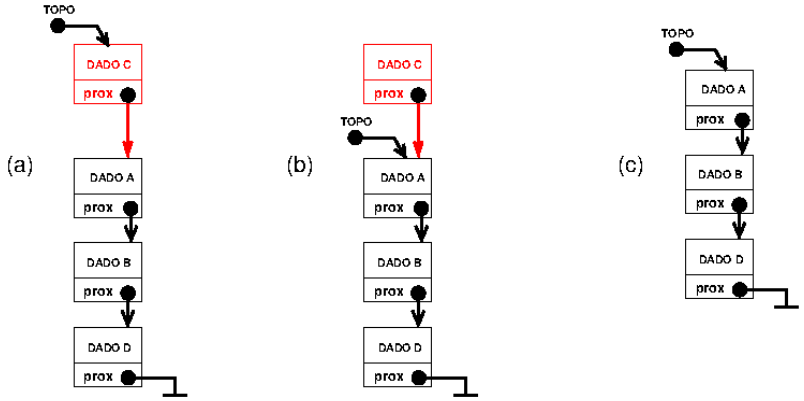

Figura 9.27: Procedimento para remoção de um elemento em uma pilha.

função.

```
1   Pilha *Desempilha(Pilha **topo)
2   // retorna endereço do elemento no topo da pilha
3   {Pilha *elem;
4
5     if (! *topo) // Pilha está vazia
6         return (NULL);
7     elem = (*topo)->prox;
8     (*topo)->prox = elem->prox;
9     return (elem);
10   }
```

Figura 9.28: Remoção do topo em uma pilha.

9.6.3 Métodos de Busca

Assim como ocorre com as filas, a busca por um elemento em uma pilha é algo estranho. Por definição os acessos em uma pilha são feitos sempre sobre o elemento topo da pilha, o que implica em não ser possível alterar diretamente os demais elementos.

No caso de ser necessário procurar um determinado elemento da pilha, o procedimento a ser usado é o de desempilhar seus elementos, empilhando-os em uma pilha auxiliar, até que se chegue ao elemento procurado. Ao encontrar o elemento procurado se deve realizar as operações desejadas e, ao final dessas operações, reempilhar os elementos que foram transferidos para a pilha auxiliar.

Exercícios

1. O processo de conversão de um número decimal para um binário pode ser feito pelo empilhamento e desempilhamento dos valores de resto da divisão por 2. Modifique as funções *Empilha* e *Desempilha* vistos na última seção para realizar essa operação.

2. Uma impressora compartilhada atende as requisições de impressão de modo FIFO. Implemente um programa que atenda tais requisições.

3. Considere um *buffer* circular com número fixo de posições (K), em que ocorram aleatoriamente operações de inserção e leitura de seus dados. Escreva as funções para essas operações considerando que as K posições foram alocadas usando estruturas dinâmicas (e não um vetor). Escreva também a função que gera as posições do *buffer*.

Capítulo 10

Métodos de ordenação

No capítulo anterior foram examinadas estruturas de dados na forma de listas. Lá pudemos ver que algumas formas de listas possuem restrições fortes na ordem dos dados, como por exemplo as filas, em que os dados na estrutura aparecem exatamente na ordem em que eles foram inseridos. Vimos também que a forma mais geral de lista permite acesso em qualquer ponto, tanto para inserção quanto para remoção. Estas listas são de grande interesse na computação por permitirem a construção de estruturas de armazenamento bastante versáteis. O grande problema desse tipo de lista é a dificuldade para a busca por determinados elementos possivelmente armazenados nela, uma vez que os elementos da lista não possuem, em princípio, nenhuma relação de ordem entre si.

Para resolver esse problema (como já adiantamos no capítulo 7) surgem as listas ordenadas. Listas ordenadas são, na realidade, listas comuns em que se rearranjam seus elementos para criar uma disposição de ordem. Elas são, por exemplo, o princípio básico da construção de bancos de dados, observando-se que esses possuem estruturas de dados bem mais complexas e necessitam de operadores mais eficientes para o acesso a informações armazenadas em discos.

Em estruturas dinâmicas o processo de criação da lista ordenada é mais simples, visto que podemos inserir o elemento em qualquer posição da lista. Para listas estáticas esse processo também não é complicado, bastando usar um campo de indexação para manter a lista ordenada, mesmo que elementos sejam inseridos ou removidos. Entretanto, muitas vezes se quer apenas armazenar dados de modo mais simples, sem o uso de indexadores para manter a relação entre os elementos, o que pode ser feito com vetores puros[1], por exemplo.

Para tais estruturas não é conveniente garantir a ordenação durante operações de inserção ou remoção. Isso ocorre pois nesse caso essas operações implicariam

[1]Por vetor puro se entende um sem índices que formem listas estáticas, isto é, um vetor em que a posição relativa do elemento na lista corresponda ao índice de sua posição no vetor

em movimentar um conjunto possivelmente grande de elementos no vetor. Assim é preferível inserir todos os elementos primeiro e depois ordená-los de uma única vez. Esse processo de ordenação pode ser feito usando algoritmos diversos, dos quais estudaremos alguns neste capítulo.

No exame desses algoritmos consideraremos, por simplicidade, que os vetores estarão ordenados de forma crescente, contendo valores inteiros. Isso, entretanto, não tira a generalidade das soluções apresentadas, pois o que muda é apenas o valor a ser comparado. Para cada algoritmo faremos um breve exame do seu custo computacional, ou seja, do tempo necessário para sua execução e como esse tempo se comporta com o crescimento do tamanho da lista. Terminaremos o capítulo com o exame de algumas formas de busca em vetores ordenados, mostrando que a ordenação melhora o desempenho da tarefa de busca.

10.1 Listas e vetores ordenados

Estruturas de dados ordenados podem ser construídas tanto na forma de lista como na forma de um vetor puro. A ordenação de vetores puros pode ocorrer a qualquer instante, usando algum dos métodos que veremos no restante desse capítulo. Já a ordenação de uma lista, quer com estrutura dinâmica quer com estrutura estática, é mais eficiente se feita durante a inserção de seus elementos, uma vez que o custo computacional de se fazer a ordenação posterior é bastante alto.

A ordenação de uma lista criada usando estruturas dinâmicas ou estáticas é feita, como indicado, a cada inserção de um elemento na lista. Assim, para a criação de uma lista ordenada basta fazer com que o procedimento visto na Figura 9.6 identifique a posição de inserção a partir do critério de ordenação escolhido. Isso é feito por laços equivalentes ao apresentado a seguir, no caso de estrutura dinâmica:

```
while (! fim_da_lista && elem_inserido < elem_atual_lista)
      elem_atual_lista =  elem_atual_lista->prox;
```

Como o procedimento de remoção não modifica a posição relativa entre os elementos restantes da lista, uma lista criada dessa forma estará sempre ordenada. Se a inserção não mantiver a ordenação entre os elementos, o processo de ordenação posterior será bastante complicado, uma vez que para a ordenação é preciso trocar dois elementos de posição na lista muitas vezes. Como o processo de troca de posição envolve a redefinição de vários ponteiros (ou índices do vetor) a cada troca, o processo de ordenar uma lista posteriormente às inserções se torna ineficiente.

Quando se usa a estrutura de um vetor puro para armazenar os dados ordenados não é necessário trocar índices/ponteiros para posicionamento relativo. O que se faz é trocar o conteúdo de cada posição do vetor, de forma a garantir que posições relativas no vetor correspondam às posições relativas entre os elementos. Existem dezenas de métodos para ordenação de vetores propostos ao longo do tempo, alguns mais simples outros mais eficientes.

Esses métodos de ordenação podem ser categorizados de vários modos. Um primeiro considera a necessidade ou não de maior espaço de memória, isto é, se para ordenar um vetor precisamos ou não de outro vetor de mesmo tamanho. Um segundo modo de categorização trata de necessidade ou não de se comparar elementos do vetor entre si. Como veremos, grande parte dos algoritmos demanda que se faça a comparação entre elementos do vetor. Nas próximas seções examinaremos alguns dos métodos baseados em comparação entre elementos do vetor, escolhidos por serem mais simples (seleção, inserção e bolha) ou mais eficientes (Shell, por heap, Quicksort e por intercalação), além de um método baseado em classificação, que não compara elementos entre si (bucket sort).

10.1.1 Comparando algoritmos

Antes de examinar os algoritmos indicados é preciso definir critérios que possam ser usados como métrica de comparação. O critério mais universal de comparação é o que chamamos de complexidade de tempo. Essa complexidade é uma medida da eficiência do algoritmo em relação ao seu tempo de execução, sendo que o quadro apresentado na página 208 resume como ela seria determinada.

Uma outra medida importante é a complexidade de espaço, que mede quanto espaço em memória o algoritmo necessita para sua operação. Essa medida foi muito importante no começo da computação, quando os sistemas tinham espaço em memória muito restrito. Atualmente isso ainda é importante para problemas que envolvam vetores efetivamente grandes, ocupando centenas de megabytes de espaço. Ocorre, entretanto, que para essa quantidade de dados também se usa estruturas de dados mais eficientes do que vetores.

Por fim, uma última métrica de comparação diz respeito à estabilidade do algoritmo. Um algoritmo de ordenação é considerado estável caso mantenha a posição relativa entre dois elementos de mesmo valor. Na prática isso significa que se um elemento A, com valor k, estava na posição i do vetor, enquanto o elemento B, também com valor k estava na posição j, com $i < j$, antes da ordenação, então eles aparecerão respectivamente nas posições m e $m + n$ após a ordenação, sendo n o número de outros elementos de valor k, menos um, que estavam entre A e B no vetor desordenado. A importância de algoritmos estáveis está em aplicações que necessitem uma primeira ordenação por uma chave e uma segunda ordenação

por uma segunda chave, como por exemplo ordenar alunos pelo nome e depois pelo ano de ingresso no curso.

10.2 Ordenação por seleção

A ordenação por seleção (*selection sort*) é um método bastante simples de ordenação. Seu princípio de funcionamento é considerar que a cada iteração o vetor pode ser dividido em duas partes, uma já ordenada e outra não ordenada. Assim, a cada iteração se seleciona o elemento de menor valor armazenado na parte não ordenada do vetor, considerando que a ordenação será crescente, e colocando ele como novo elemento ao final da parte ordenada. O procedimento continua, agora considerando que a parte não ordenada começa na posição seguinte do vetor, até que todos os elementos tenham sido escolhidos pelo algoritmo.

A Figura 10.1 apresenta esses passos para um pequeno vetor. Na primeira linha o vetor aparece com uma parte já ordenada (os primeiros dois elementos, marcados pelo retângulo azul) e uma ainda não ordenada. Na parte não ordenada se seleciona o menor valor (8, circulado em vermelho). Troca-se então o 8 e o 12 de posição, crescendo a parte ordenada do vetor, como visto na segunda linha. Na iteração seguinte se coloca o 9 na parte ordenada, trocando de posição com o 10. A última iteração mostrada faz a troca do 10 pelo 11.

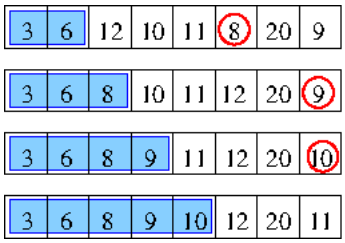

Figura 10.1: Movimentações no vetor durante a ordenação por seleção.

10.2.1 Projeto e implementação

A ordenação por seleção pode ser implementada de modo bastante simples. Da descrição apresentada é possível ver que o procedimento é composto de duas partes, uma que encontra o menor valor entre os elementos ainda não ordenados do vetor e outra que vai compondo esses elementos na parte ordenada, até ter todos os elementos ordenados.

As duas partes podem ser construídas com laços de repetição, sendo um para acrescentar elementos no final do trecho ordenado do vetor e outro, interno ao

primeiro, para encontrar o menor valor no trecho não ordenado. Ao final do laço mais interno ocorre, quando necessário, a troca do elemento que ocupava a posição inicial da parte não ordenada com o elemento de menor valor.

O código apresentado na Figura 10.2 implementa a ordenação por seleção por valores crescentes. Nele se observa que o laço do primeiro `for` (linhas 4 a 14) percorre o vetor até a penúltima posição, considerando que ao ter essas posições ordenadas o elemento restante será, automaticamente, o de maior valor. Já o segundo laço (linhas 6 a 8) busca pelo menor valor na parte ainda não ordenada do vetor. que é colocado em sua posição correta nas linhas 10 a 12.

```
1   void selecao(int vetor[N]) // N é declarada como global
2   {int i, j, menor, aux;
3
4     for (i=0; i<N-1; i++)
5     { menor = i;
6       for (j=i+1; j<N; j++)
7           if (vetor[j] < vetor[menor])
8               menor = j;
9       if (menor != i)
10           { aux = vetor[i];
11             vetor[i] = vetor[menor];
12             vetor[menor] = aux;
13           }
14    }
15    return;
16  }
```

Figura 10.2: Ordenação de vetor pelo algoritmo de seleção.

Devemos observar aqui que esse algoritmo pode ser considerado estável, pois em sua implementação é possível garantir que a posição relativa entre elementos de igual valor é preservada após a ordenação. Isso ocorre pois o teste na linha 7 não troca o menor elemento caso os valores sejam iguais.

O que é análise de complexidade -
Análise de complexidade é um tópico de teoria da computação que busca determinar, a partir do algoritmo aplicado em uma solução computacional, o comportamento do algoritmo quando se muda o tamanho do problema. Isso significa, por exemplo, identificar o quanto aumento o tempo de execução de um programa se o volume de dados dobrar. O estudo formal de complexidade não cabe neste texto, porém é valioso no momento de comparar a eficiência dos algoritmos de ordenação aqui estudados.
De modo simplificado podemos dizer que a complexidade de um algoritmo é dada pela notação $\mathcal{O}(f(n))$. A função $f(n)$ dentro dos parenteses indica como se comporta o tempo de execução do algoritmo em relação ao tamanho n dos dados. Por exemplo, $\mathcal{O}(n^2)$ indica que o tempo de execução do programa aumenta quadraticamente com o tamanho dos dados. Existem regras bem definidas para a determinação da complexidade a partir da contagem das instruções a serem executadas para um conjunto de dados de tamanho n, mas aqui apresentaremos apenas sua versão simplificada.

10.2.2 Análise de complexidade

Para determinar a complexidade (ver quadro acima) do algoritmo de ordenação por seleção "contamos" quantas instruções são executadas para fazer a ordenação. Para essa contagem são desconsideradas as instruções que não aumentam o tempo de execução. Por exemplo, a menos de laços aninhados, as instruções dentro de um laço são contabilizadas como se fossem uma só, pois todas são executadas no máximo uma vez a cada iteração. Já o número de iterações é influenciado pelo tamanho do problema.

De maneira simplificada temos apenas que contar quantas vezes executamos os laços existentes no algoritmo.Nele temos dois laços de repetição aninhados. O laço exterior é executado $n - 1$ vezes, como se percebe do comando na linha 4. Isso resultaria numa complexidade de $\mathcal{O}(n)$ se não existisse o laço interno.

Para determinar a contribuição do laço interno (linhas 6 a 8) temos que a cada iteração do laço externo ele é executado $n - i$ vezes. Para evitar uma formulação muito complexa o que se faz aqui é considerar que será executado n vezes, o que é uma aproximação razoável. Assim, para cada iteração externa temos n iterações internas, resultando numa complexidade de $\mathcal{O}(n \times n)$ ou $\mathcal{O}(n^2)$.

De modo prático, isso significa que se gastamos t unidades de tempo para ordenar um vetor de tamanho n, gastaremos $4t$ unidades de tempo para ordenar um vetor de tamanho $2n$. Esse não é um bom resultado, pois significa um aumento quadrático no tempo consumido para ordenar um vetor que dobre de tamanho.

10.3 Ordenação por inserção

Ordenação por inserção é um algoritmo que imita o que a maioria das pessoas faz para ordenar um conjunto de cartas de baralho na mão. Em geral, o que se faz é buscar a posição relativa da carta entre as cartas já ordenadas, inserindo a carta numa posição em que ela será maior que as cartas anteriores e menor que a carta imediatamente posterior (se ela não permanecer em sua posição atua). Nesse processo, todas as cartas maiores são movidas uma posição para cima na mão. Esse procedimento é repetido até que a última carta na mão seja inserida em sua posição correta.

A Figura 10.3 mostra parte do processo de ordenação por inserção. Os valores circulados são aqueles que serão "inseridos" em sua posição relativa a cada passo. Em cada linha da figura não estão representadas as movimentações intermediárias em cada iteração. Assim, na primeira iteração se leva o 6 até a primeira posição (trocando com o 12). Na segunda iteração se leva o 8 até a posição posterior ao 6, pois assume-se que se existirem outras posições anteriores àquela ocupada pelo 6, então elas conterão valores menores que o 6.

O processo continua, agora levando o 3 até a primeira posição do vetor, trocando de posição sucessivamente com o 12, o 8 e o 6. A última linha apenas mostra que o próximo valor a ser colocado em sua posição relativa é o 11, o que significaria apenas trocar de posição com o 12, uma vez que as posições anteriores contém valores menores que o 11.

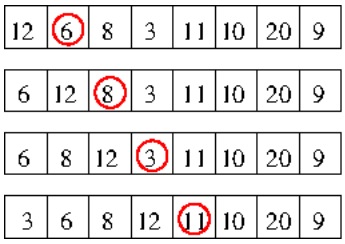

Figura 10.3: Movimentações no vetor durante a ordenação por inserção.

Devemos observar aqui que a ordenação por inserção apenas garante que um valor está em sua posição definitiva ao final do algoritmo. Antes disso, o que se pode garantir é que os valores armazenados nas posições anteriores à posição atual do valor a ser inserido estão relativamente ordenados entre si. Isso significa que os valores ainda não ordenados poderão, no máximo, alterar a distância entre dois valores, mas não a ordem entre eles.

10.3.1 Projeto e implementação

A ordenação por inserção também pode ser implementada de modo bastante simples. O que se tem que fazer também é separar o vetor em duas partes, uma com valores já inseridos em sua ordem relativa e outra com valores ainda a serem inseridos. A cada iteração o primeiro elemento da segunda parte é o elemento que será inserido em sua posição relativa. Essa posição é definida como a posterior ao primeiro elemento da parte inicial do vetor que seja menor ou igual ao elemento sendo inserido.

Esse processo pode ser construído com dois laços aninhados. O laço mais externo percorre o vetor a partir de sua segunda posição até a última, podendo ser implementado com um comando `for`. Já o segundo laço envolve a busca pela posição relativa do elemento sendo inserido, o que é mais facilmente realizado com um comando `while`, que tem sua terminação determinada por se chegar a um elemento menor do que o inserido ou na primeira posição do vetor.

O código apresentado na Figura 10.4 implementa a ordenação por inserção. O laço exterior, que é controlado pelo comando `for` da linha 4, percorre o vetor a partir de seu segundo elemento, buscando para cada elemento a sua posição relativa no vetor. O corpo desse laço é composto por três partes: o armazenamento do elemento que será inserido em uma variável auxiliar, o laço que busca sua posição relativa e o posicionamento efetivo desse elemento no vetor.

```
1   void insercao(int vetor[N]) // N é declarada como global
2   {int i, j, valor;
3
4     for (i=1; i<N; i++)
5     { valor = vetor[i];
6       j = i;
7       while (j>0 && vetor[j-1] > valor)
8       { vetor[j] = vetor[j-1];
9         j--;
10       }
11       if (j != i)           // É possível omitir este teste, apenas
12           vetor[j] = valor; // fazendo a atribuição todas as vezes
13     }
14     return;
15   }
```

Figura 10.4: Ordenação de vetor pelo algoritmo de inserção.

O laço interno é controlado pelo comando `while` da linha 7. Seu corpo basicamente vai mudando os elementos maiores que o elemento sendo inserido uma

posição para cima no vetor. O laço para quando o elemento comparado é menor ou igual ao valor inserido ou, no limite, atinge-se a primeira posição do vetor. Por fim, a atribuição dentro do comando `if` (linhas 11 e 12) coloca o valor inserido em sua posição relativa na parte ordenada do vetor. Como aparece comentado, o teste realizado apenas evita que se copie o valor avaliado sobre ele mesmo, o que pode ocorrer quando o laço interno não for percorrido pelo menos uma vez. Se considerarmos vetores muito grandes, é preferível fazer a atribuição sempre, economizando testes que retornariam verdade quase sempre.

É fácil de perceber, pelo teste na linha 7, que esse algoritmo também é estável, preservando a posição relativa entre os elementos originais do vetor.

10.3.2 Análise de complexidade

A ordenação por inserção também é composta por dois laços aninhados. O primeiro deles é executado $n - 1$ vezes para um vetor de tamanho n. Já o laço interno é executado um número indeterminado de vezes, pois o teste no comando `while` depende de como o vetor está originalmente arranjado.

No pior dos casos esse laço será executado i vezes, sendo i o índice da posição do elemento que vai ser inserido no vetor. Essa posição varia de 1 até n, o que leva a uma situação semelhante ao visto para a ordenação por seleção. Isso implica, portanto, que a complexidade do algoritmo será de $\mathcal{O}(n^2)$. Na prática, entretanto, a ordenação por inserção apresenta um comportamento bem melhor do que o de seleção, uma vez que na ordenação por seleção os laços são executados integralmente, enquanto na inserção a execução do laço interno pode ser interrompida bem antes de i iterações.

10.4 Ordenação pelo método bolha

A ideia básica do método bolha é fazer com que valores mais altos do vetor sejam sucessivamente levados para posições mais próximas do final do vetor. O nome do algoritmo vem exatamente da visão de que bolhas dentro de um líquido sobem para a superfície, que é o que ocorre com os valores mais altos dentro do vetor.

Num primeiro momento pode-se pensar que ele atua de modo inverso ao algoritmo de seleção, em que um leva os menores valores para o começo e o outro os maiores valores para o final. A diferença entre eles é a forma como isso é feito. Na ordenação por seleção a cada iteração temos os menores valores efetivamente na parte inicial do vetor. Já na ordenação por bolha um valor maior "sobe" no vetor até encontrar um valor maior que ele, quando é parado para que esse valor maior suba primeiro. Esse processo é repetido até que nenhum valor (bolha) suba durante uma iteração.

Parte da aplicação deste algoritmo em um pequeno vetor é vista na Figura 10.5. Os valores circulados em cada linha representam o valor que será borbulhado para o final do vetor. Num primeiro momento o valor 12 será levado até a posição imediatamente anterior ao ocupado pelo 20. Como 20 é maior que 12, então ele passa a ocupar a bolha que caminha para o final do vetor (segunda linha). A segunda iteração do algoritmo começa na terceira linha da figura, agora com o 6 como bolha. Como o valor seguinte já é maior que ele, então ele é mantido no lugar e o 8 passa a ocupar a bolha.

Dando continuidade nessa iteração a bolha sobe até o valor 11, que por ser maior que o 8 passa a ocupar a bolha. Se fossemos continuar a sua aplicação, o 11 trocaria de lugar com o 10 e passaria a vez para o 12, que trocaria de lugar com o 9, encerrando a segunda iteração. Como dito antes, nas próximas iterações esse processo é repetido, até que não ocorra nenhuma troca de posição entre os elementos do vetor, o que ocorre quando o vetor estiver ordenado.

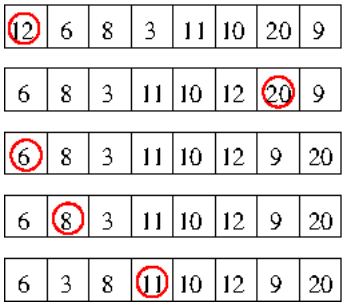

Figura 10.5: Movimentações no vetor durante a ordenação pelo método bolha.

10.4.1 Projeto e implementação

Da descrição feita do algoritmo se percebe que temos mais uma vez dois laços de repetição. Um que repete o processo de borbulhamento até que não ocorram mais trocas e outro que percorre o vetor fazendo a bolha subir. O laço externo tem sua terminação definida pela ocorrência ou não de uma troca de posição entre elementos do vetor (comando `while` da linha 4). O corpo desse laço conterá o laço interno e a atribuição de valor para a última posição da bolha (linha 17).

Como o laço interno deverá percorrer o vetor todo, independente de haver ou não trocas de posição, pode ser implementado com um comando `for`. O corpo desse laço é composto por um único teste, que decide se o elemento testado continuará a ser movido para o fim do vetor ou se um outro elemento passará a ocupar a bolha. O primeiro caso está nas linhas 9 e 10, em que se observa a marcação da ocorrência de uma troca de posição. A mudança do elemento da

bolha ocorre nas linhas 13 e 14.

```
1   void bolha(int vetor[N]) // N é declarada como variável global
2   {int i, j=N, valor, trocou=1;
3
4     while (trocou)
5     { valor = vetor[0];
6       trocou = 0;
7       for (i=1; i<j; i++)
8       { if (vetor[i] < valor)
9             { vetor[i-1] = vetor[i];
10              trocou = 1;
11            }
12          else
13            { vetor[i-1] = valor;
14              valor = vetor[i];
15            }
16      }
17      vetor[i-1] = valor;     --j;
18    }
19    return;
20  }
```

Figura 10.6: Ordenação de vetor pelo método bolha.

Vale observar aqui que como dois elementos trocam de posição apenas quando elemento da posição i é menor que o elemento da bolha, então o algoritmo também é estável.

10.4.2 Análise de complexidade

Dos três algoritmos já examinados fica evidente que o método bolha é o que envolve mais operações para sua execução. Isso se reflete na prática com o fato dele ter tempo de execução maior do que os algoritmos de inserção e seleção. Do ponto de vista teórico, entretanto, os três apresentam comportamentos equivalentes, isto é, todos têm comportamento quadrático em relação ao aumento do tamanho do vetor.

Isso pode ser visto, no caso do bolha, verificando que o laço interno é repetido $n-1$ vezes para cada iteração do laço externo. Já este laço será repetido até que não se faça uma troca de valores no laço interno, o que depende de como o vetor está originalmente organizado. A pior organização possível implica na execução desse laço $n+1$ vezes (a passagem adicional é para confirmar que não houve mais

trocas de posição). Isso resulta, portanto, em uma complexidade de $\mathcal{O}(n^2)$. O melhor caso ocorre quando o vetor já está ordenado, com complexidade $\mathcal{O}(n)$. O caso geral acaba sendo reduzido a $\mathcal{O}(n^2)$.

Nunca faça ordenação pelo método bolha -
O método bolha é o pior dos métodos de ordenação aqui vistos. Embora tenha ordem de complexidade semelhante ao dos métodos de inserção e seleção, ele é bem mais lento do que esses dois e, como será visto a seguir, muitíssimo mais lento do que métodos como Shell sort, quicksort e vários outros.
Assim, ele foi apresentado aqui apenas para que se possa ver que existem soluções inerentemente ruins. Nunca ordene seus dados usando o método bolha. Caso sejam poucos elementos no vetor (até 10 mil elementos, por ex.), use o método de inserção ou mesmo o de seleção, que são algoritmos também bastante simples de se implementar e mais rápidos do que o bolha.

10.5 Ordenação por Shell sort

O algoritmo de ordenação proposto por Donald Shell em 1959 procura ordenar o vetor considerando que é mais simples ordenar vetores pequenos do que vetores grandes. Nele o que se faz é partir de vários subvetores do vetor original, que são ordenados usando o algoritmo de inserção. Esse conjunto de subvetores é então reduzido por um fator preestabelecido, sendo que a cada novo conjunto de vetores se refaz a ordenação por inserção. Esse procedimento continua até que se tenha apenas um vetor.

Considerando que o algoritmo de ordenação por inserção tem complexidade de $\mathcal{O}(n^2)$ no pior caso e $\mathcal{O}(n)$ no melhor caso, o Shell sort se torna eficiente pois começa com vetores bem pequenos quando existe maior desorganização. A partir de ordenações intermediárias se aumenta o tamanho dos vetores, porém estes estarão mais organizados e próximos do melhor caso para a aplicação do algoritmo de inserção.

Além disso, o algoritmo dispensa a necessidade de armazenamento adicional pois os k subvetores serão, na prática, vetores virtuais com elementos distantes k posições entre si (o passo do algoritmo). Assim, para um vetor com digamos 15 elementos, caso sejam criados inicialmente 4 vetores, teríamos o vetor A com os elementos das posições 0, 4, 8 e 12 do vetor original, mais o vetor B com as posições 1, 5, 9 e 13, o vetor C com as posições 2, 6, 10 e 14 e, finalmente, o vetor D com os elementos das posições 3, 7 e 11.

Esses vetores virtuais são ordenados usando o algoritmo de inserção, embora na prática qualquer outro algoritmo simples possa ser aplicado. Após uma primeira ordenação se diminui o tamanho do passo (e de vetores virtuais), refazendo-se a ordenação desses novos vetores. O processo de redução do passo segue até que se chegue ao tamanho de uma posição, quando o vetor é ordenado pela última vez. A vantagem desse processo é que a cada passo os valores iniciais armazenados caminham mais rapidamente para a posição que deverão ficar após a ordenação, mesmo para vetores grandes.

Esse processo é ilustrado na Figura 10.7, com um vetor de 10 posições e passo inicial igual a 4. Na primeira iteração temos vetores com os seguintes elementos: (12, 3, 10), (6, 14, 9), (11, 20) e (11, 8). Esses vetores são ordenados, resultando no vetor da segunda linha da figura. Nele se aplica uma nova iteração, agora com passo igual a 2, com a ordenação dos vetores (3, 11, 10, 20, 12) e (6, 8, 9, 11, 12). Após ordenar esses vetores chega-se ao vetor da terceira linha, ao qual se aplica a ordenação final pois se chegou ao passo de tamanho 1.

Apenas a título de ilustração, a aplicação da ordenação por inserção no vetor inicial resultaria em 23 trocas de posição dentro do vetor. Já a aplicação do Shell sort resulta em 11 trocas no total, com poucas trocas na aplicação do algoritmo de inserção a cada fase.

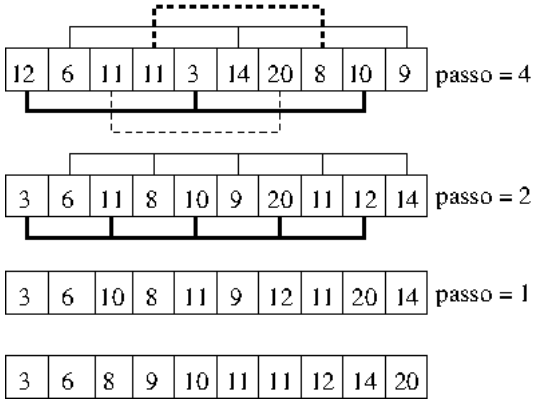

Figura 10.7: Movimentações no vetor durante a ordenação pelo Shell sort.

10.5.1 Projeto e implementação

A ordenação por Shell sort envolve basicamente a aplicação repetida de um algoritmo de ordenação mais simples. Para o seu projeto falta definir qual é o tamanho do passo inicial e qual a razão de redução do passo a cada iteração. Desde a sua formulação em 1959 foram propostas diversas formas de determinar esses valores,

sempre procurando melhorar a eficiência da ordenação. Na prática o processo deve envolver uma quantidade relativamente pequena de passos, evitando uma quantidade excessiva de execuções do algoritmo de ordenação básico.

A proposta original do algoritmo definia os seguintes parâmetros:

Passo inicial: $\lfloor \frac{N}{2} \rfloor$

Sequência de passos: $\lfloor \frac{N}{2} \rfloor$, $\lfloor \frac{N}{4} \rfloor$, ..., 1

Das várias formas de definição de passo apresentadas, uma das que oferece melhor desempenho e simplicidade é a proposta por Donald Knuth em 1973, a partir da proposta feita por Pratt dois anos antes. Na formulação de Knuth temos que o termo geral é dado por:

Passo inicial: $\frac{3^k-1}{2} < \lceil \frac{N}{2} \rceil$

Sequência de passos: $\frac{3^k-1}{2}$, ..., 121, 40, 13, 4, 1

Isso facilita a implementação do algoritmo pois para determinar o valor inicial basta o laço a seguir:

```
k = 1;
while (k < N/2)
    k = 3*k + 1;
```

Sendo que a redução no tamanho do passo, durante as várias fases do algoritmo, é dada por `k = k/3`, uma vez que a divisão de inteiros trunca o resultado. Assim, a implementação proposta por Knuth pode ser vista no código apresentado na Figura 10.8.

Nesse código se observa que nas linhas 5 e 7 temos a determinação do passo inicial do algoritmo. O laço controlado pelo comando `while` da linha 6 representa a aplicação de cada passo do Shell sort, sendo o tamanho do passo determinado pela atribuição da linha 7. O comando `for` da linha 8 controla a aplicação da ordenação para os k vetores parciais, sendo que a ordenação de cada um deles ocorre no laço iniciado na linha 9.

Devemos observar aqui que embora a aplicação do algoritmo de inserção garanta estabilidade num dado vetor, a separação realizada em vetores "intercalados" faz com que o Shell sort não seja um algoritmo estável.

10.5.2 Análise de complexidade

A análise de complexidade do Shell sort é dependente da forma como são gerados os passos para sua implementação. Em sua formulação original a complexidade de

```
1   void shellsort(int vetor[N]) // N é declarada como variável global
2   {int i, j, k, l, valor;
3
4     k = 1;
5     while (k < N/2)  k = 3*k + 1;  // calculando passo inicial
6     while (k != 1)
7     { k = k/3; // acertando o valor do passo para iteração atual
8       for (l=0; l<k; l++) // laço para executar insertion k vezes
9       { for (i=l+k; i<N; i=i+k) // algoritmo insertion ajustado
10        { valor = vetor[i];
11          j = i;
12          while (j-k>=0 && vetor[j-k] > valor)
13          { vetor[j] = vetor[j-k];
14            j=j-k;
15          }
16          vetor[j] = valor;
17        }
18       }
19     }
20     return;
21  }
```

Figura 10.8: Ordenação de vetor pelo método Shell sort.

pior caso é $\mathcal{O}(n^2)$. Na formulação de Knuth apresentada na Figura 10.8 esse valor cai para $n^{1.5}$. Existem propostas de determinação de passo que ainda reduzem esse valor, porém com formulação mais complicada para determinar o passo. Por exemplo, Sedgewick, em 1982 propôs a seguinte sequência, com pior caso igual a $\mathcal{O}(n^{\frac{4}{3}})$

Passo inicial: $P = 4^k + 3 \,.\, 2^{k-1} + 1$, tal que $P < \frac{N}{2}$ e $k \geq 1$

Sequência de passos: ..., 281, 77, 23, 8, 1

De forma geral a complexidade do Shell sort é maior que $n \log n$, mas com valores bem melhores do que os algoritmos anteriores. Isso é interessante se lembrarmos que sua implementação faz uso do algoritmo de inserção.

10.6 Ordenação por heapsort

Este algoritmo trabalha em duas fases. Numa primeira fase os valores contidos no vetor são reorganizados logicamente dentro dele de forma a gerar um *heap*, que é

basicamente uma estrutura de árvore binária. A estrutura de *heap* faz com que os elementos do vetor sejam separados em "camadas" de valores, de tal forma que na camada superior esteja o elemento de maior valor. Devemos observar aqui que embora se crie uma estrutura de árvore binária, ela é criada usando as posições do próprio vetor e não uma estrutura separada. Isso é possível pois podemos caracterizar facilmente os filhos de um elemento que esteja na posição i da árvore como os ocupantes das posições $2.i + 1$ e $2.i + 2$.

Na segunda fase os elementos são retirados da organização de *heap* de modo a que se reorganize o vetor do seu final para o começo. Isso significa mover o maior elemento do *heap* da primeira para a última posição do vetor. Isso é feito reorganizando a estrutura do *heap* para manter sua estrutura com os elementos restantes. Isso é repetido até que o vetor esteja de fato ordenado.

A Figura 10.9 mostra como o vetor original é transformado no vetor de *heap*, além de mostrar como esses elementos estariam armazenados na árvore binária. Como se pode observar, a cada nível da árvore os filhos são menores que o nó pai. Com isso o elemento na raiz da árvore será sempre o maior elemento que ainda não tenha sido ordenado. Com essa informação o processo de ordenação leva sempre o elemento que está na primeira posição do vetor para a última posição ainda não ordenada.

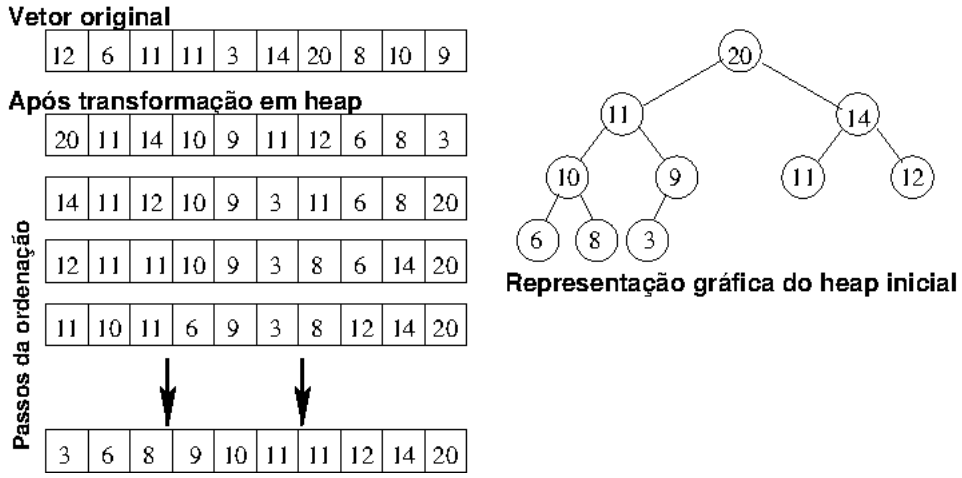

Figura 10.9: Sequência de operações no heapsort.

10.6.1 Projeto e implementação do método

A implementação do *heap sort* é separada em duas funções. A primeira delas faz a alocação dos elementos do vetor dentro do *heap* e a segunda faz a alocação dos

elementos do *heap* dentro do vetor, a partir de sua última posição. Começando então pela função que gera o *heap*, temos que o processo é relativamente simples, bastando inserir os elementos do vetor em posições hierárquicas de modo a ter nós filhos sempre menores que nós pais.

Esse processo é feito colocando o elemento i do vetor original inicialmente na posição i do *heap*. A partir disso se verifica se esse elemento é menor ou maior que o seu pai. No caso de ser maior ocorre a troca desses elementos, sendo o processo repetido até que o elemento chegue na raiz do *heap* ou seja menor que seu pai. Esse processo será aqui chamado de `criaHeap`, tendo seu código apresentado na Figura 10.10. Nesse código vale ressaltar que o processo parte do princípio de que a cada iteração se leva valores maiores para mais próximos do topo do *heap*, sabendo que é preciso apenas considerar o caminho para cima e não dois caminhos para baixo, caso considerássemos levar valores menores para a parte de baixo.

No código da Figura 10.10 o processo de trazer valores maiores para o topo do *heap* ocorre no laço das linhas 6 a 8. A cada iteração o pai do elemento na posição f examinado está na camada acima, tendo por índice o valor $(f-1)/2$. Essa relação corresponde exatamente aos índices de uma árvore binária construída dentro do vetor.

```
1   void criaHeap (int vetor[N], int m)
2   {int i, f, x;
3
4     for (i = 1; i < m; ++i)
5     {  f = i;     x = vetor[f];
6        while (f > 0 && vetor[(f-1)/2] < x) {
7           vetor[f] = vetor[(f-1)/2];
8           f = (f-1)/2; }
9        vetor[f] = x;
10     }
11  }
```

Figura 10.10: Fase de criação do *heap* para ordenação por *heap sort*.

Após inserir todos os elementos no *heap* é possível passar para a segunda fase do algoritmo. Nela o que se faz é trocar o primeiro elemento do vetor com o elemento na última posição ainda não ordenada do vetor. Após a troca é necessário refazer o *heap*, trocando o elemento que foi levado para a primeira posição com seus filhos até que a condição de ordem do *heap* seja obtida. Isso é equivalente a aplicar novamente o algoritmo para a criação do *heap*, ou seja, a função `criaHeap`.

Ocorre, entretanto, que o elemento levado para a primeira posição do ve-

tor, topo do *heap*, tem valor relativamente baixo. Assim, aplicar diretamente a função `criaHeap` é ineficiente por demandar muitas trocas de elementos. O que normalmente se faz é aplicar um método que leva esse elemento para as camadas inferiores do *heap*, diminuindo a complexidade do algoritmo.

Esse método resulta no código da função `desceTopo`, apresentado na Figura 10.11. A determinação efetiva da posição a ser ocupada pelo elemento colocado no topo é realizada dentro do laço do comando `while` da linha 4. Nele se faz sucessivamente as trocas de elementos no vetor, trazendo os maiores valores para as camadas superiores na linha 7. As atribuições realizadas na linha 8 fazem a mudança de uma camada para a camada imediatamente inferior a cada iteração. Esse processo continua até chegar ao final do *heap* ou encontrar a posição correta.

```
1   void desceTopo(int vetor[N], int m)
2   {int p = 0, f = 1, x = vetor[0];
3
4     while (f <= m) {
5        if (f < m && vetor[f] < vetor[f+1]) ++f; // escolhe o ramo
6        if (x >= vetor[f]) break;  // para a descida se já chegou
7        vetor[p] = vetor[f];  // sobe um elemento de camada
8        p = f; f = 2*p+1;
9     }
10     vetor[p] = x;  // coloca o valor na posição correta
11  }
```

Figura 10.11: Acerto do *heap* a cada iteração do algoritmo.

O corpo principal do algoritmo é visto na Figura 10.12. Nele se observa a chamada inicial para a criação do *heap* inicial. Depois disso temos um laço que vai levando o primeiro elemento do *heap* para o final da parte ainda não ordenada, nos comandos de troca de valores (linhas 6 a 8). Depois é chamada a função de acerto do *heap*, sempre considerando que mais um elemento está ordenado no final do vetor.

É fácil observar que este algoritmo também não é estável. A aplicação da função `criaHeap` não garante a manutenção de ordem entre elementos de mesmo valor, pois esses elementos provavelmente serão colocados em "ramos" diferentes da árvore de *heap*. Além disso, as várias aplicações da função `desceTopo` também não garante a preservação de ordem entre elementos de mesmo valor.

```
1  void heapsort (int n, int vetor[N])
2  {int m, aux;
3
4     criaHeap (vetor, n);
5     for (m = n-1; m > 1; --m) {
6        aux = vetor[0];
7        vetor[0] = vetor[m];
8        vetor[m] = aux;
9        desceTopo (vetor, m-1);
10    }
11 }
```

Figura 10.12: Fase de ordenação a partir do *heap* criado.

10.6.2 Análise de complexidade

A complexidade do heapsort pode ser avaliada determinando a complexidade das duas fases do algoritmo. Começando então pela análise da função `criaHeap`, vemos que ela é composta por dois laços aninhados. O laço `for` é executado $n-1$ vezes para um vetor de tamanho n. Já o laço `while` é executado $\log i$ vezes para cada valor de i do laço `for`. Assim, a complexidade da primeira fase é, no pior caso, da ordem de $\mathcal{O}(n \log n)$.

Para a segunda fase vemos que temos também um laço `for` com n iterações, sendo que para cada iteração temos que determinar a complexidade da função `desceTopo`. Esta função, por sua vez, tem um laço `while` que é executado no máximo $\log m$ vezes. Desse modo a complexidade da segunda fase será também da ordem de $\mathcal{O}(n \log n)$. Como as duas fases tem a mesma ordem de complexidade e são sequenciais, a ordem de complexidade do algoritmo será também $\mathcal{O}(n \log n)$, uma vez que do ponto de vista teórico $2n$ e n são diretamente proporcionais ao crescimento de n.

10.7 Ordenação por Quicksort

Quicksort é um algoritmo de ordenação proposto por Tony Hoare em 1960, como um método para organizar um dicionário. O algoritmo tem por base o fato de ser relativamente simples separar os elementos do vetor em dois conjuntos, um com valores maiores que um dado elemento, chamado pivô, e outro com valores menores que o pivô. Essa separação pode ser continuamente aplicada a cada subconjunto formado, até que se tenha todos os elementos do vetor ordenados.

Esse processo é apresentado na Figura 10.13, em que a primeira linha apre-

senta o vetor original. Escolhendo-se o primeiro elemento do vetor como pivô, os demais elementos são sucessivamente trocados de posição até que todos os valores maiores que o pivô estejam na parte final do vetor e os menores em seu começo. Nesse ponto, ocorre a troca de posição do elemento pivô (4), como mostrado no vetor em azul.

4	9	8	2	3	5	1	8	6	7
4	1	8	2	3	5	9	8	6	7
4	1	3	2	8	5	9	8	6	7
2	1	3	4	8	5	9	8	6	7

Figura 10.13: Sequência inicial de operações no Quicksort.

Isso resulta em termos um subvetor com os elementos anteriores à posição final do pivô e outro com os elementos posteriores à essa posição. Nesse momento se aplica novamente o algoritmo para cada um dos subvetores, em um processo recursivo até que os vetores criados tenham no máximo um elemento, quando estariam obviamente ordenados.

O vetor em vermelho apresenta o resultado da aplicação do algoritmo na parte inicial do vetor (valores menores que o pivô inicial). Na sequência aparecem as movimentações executadas para a parte final do vetor, executadas recursivamente, separando-se cada reaplicação do algoritmo com linhas horizontais.

1	2	3	4	8	5	9	8	6	7
1	2	3	4	8	5	7	8	6	9
1	2	3	4	8	5	7	6	8	9
1	2	3	4	6	5	7	8	8	9
1	2	3	4	5	6	7	8	8	9

Figura 10.14: Sequência de operações no Quicksort.

Esse procedimento é bastante eficiente, apesar das sucessivas chamadas recursivas. Apenas como ilustração, o vetor deste exemplo demandaria 21 trocas de elementos se aplicado o algoritmo de inserção, em vez das 8 trocas feitas com o Quicksort.

10.7.1 Projeto e implementação do método

A implementação do Quicksort é inerentemente recursiva. Seu projeto envolve determinar um critério de parada para a recursão, um método para identificar os trechos do vetor a serem trabalhados e a separação dos elementos do vetor.

O critério de parada consiste em terminar a recursão quando o vetor recebido tiver um ou menos elementos. Isso pode ser relaxado se vetores com menos que K elementos forem ordenados por outro algoritmo como o de inserção.

A identificação dos trechos do vetor a serem trabalhados numa chamada recursiva também é simples. Basta assumir que a cada chamada recursiva sejam passados como parâmetros os índices do primeiro e do último elemento do trecho. Assim, cada subvetor fica facilmente identificável e também se facilita o trabalho de determinar a condição de parada, que é a diferença entre esses índices.

Por fim, a separação dos elementos no vetor também é simples. Nele primeiro se define qual será o pivô. Apesar ser possível escolher um pivô que maximize a separação, isto é, um pivô que ao final separe o vetor em duas partes de mesmo tamanho, normalmente se usa uma estratégia mais simples, pois com isso se evita uma sobrecarga de processamento que eliminaria o ganho da escolha otimizada. Assim, usam-se estratégias mais simples como escolher o primeiro elemento do vetor, um elemento aleatório qualquer ou soluções como mediana de três elementos ou mediana de nove elementos (mediana de três medianas de três elementos), que são estatisticamente mais eficientes.

A implementação descrita a seguir, na Figura 10.15, considera a escolha do pivô como sendo a mediana de três elementos. Nessa implementação pega-se os elementos da primeira e última posição, além do elemento na posição intermediária. O pivô será o elemento de valor intermediário entre esses três. Uma vez definido o pivô a primeira operação coloca esse valor na última posição do vetor, para facilitar o processo de separação entre menores e maiores que o pivô.

A separação é feita em um laço em que a cada iteração se verifica se o elemento naquela posição é menor ou maior que o pivô (linha 10). Se for menor, ele é colocado numa posição mais baixa do vetor, na região que conterá os elementos menores que o pivô. Ao final desse laço troca-se o elemento maior que o pivô que esteja na posição mais baixa no vetor (bigger), com o pivô.

Com isso a nova posição ocupada pelo pivô separa o vetor com elementos menores que ele nas posições anteriores a ele, e os elementos maiores que ele nas posições posteriores, que são ordenados pela aplicação recursiva do algoritmo.

Por fim, a determinação dos limites dos novos vetores dentro do processo recursivo é feita na chamada recursiva da função `quicksort`. Uma chamada, na linha 16, é feita para o vetor que vai da atual posição inicial até a posição anterior ao pivô, contendo os elementos menores ou iguais ao pivô. Já a segunda chamada, linha 17, coloca como limites a posição posterior ao pivô e a posição

```
1  void quicksort(int *vet, int lo, int hi)
2  {int pivo, i, bigger;
3
4    if (lo < hi)  // recursão termina se vetor tem zero ou um elemento
5        { pivo = mediana3(vet, lo, (lo+hi)/2, hi);
6          troca(&vet[pivo], &vet[hi]);
7          bigger = low;
8          for (i=bigger; i < hi; i++) // laço para separar valores
9                                      // menores e maiores que o pivô
10         { if (vet[i] < vet[hi])
11               { troca(&vet[i], &vet[bigger])
12                 bigger++;
13               }
14         }
15         troca(&vet[hi], &vet[bigger]); // coloca pivô no seu lugar
16         quicksort(vet, lo, bigger-1); // recursão da parte inferior
17         quicksort(vet, bigger+1, hi); // recursão da parte superior
18       }
19 }
```

Figura 10.15: Implementação do Quicksort em C.

final do vetor de entrada, que é a parte do vetor com elementos maiores que o pivô.

Com essa implementação fica claro que o Quicksort não é um algoritmo estável. Todas as trocas de posição feitas no processo de separação de elementos maiores e menores que o pivô alteram a ordem relativa inicial entre elementos de mesmo valor, removendo a condição de estabilidade portanto.

10.7.2 Análise de complexidade

O Quicksort é um algoritmo extremamente eficiente, apesar de ter uma complexidade de pior caso alta. O resultado efetivo da complexidade depende da aleatoriedade dos elementos do vetor e também do pivô escolhido a cada iteração. Em particular, quanto mais aleatória for a distribuição de valores no vetor, melhor será o resultado do ponto de vista de eficiência da ordenação.

No pior caso a complexidade do algoritmo é da ordem de $\mathcal{O}(n^2)$, que é equivalente aos piores algoritmos vistos no início do capítulo. Entretanto, esse resultado ocorre quando a escolha pelo pivô resulta sempre no elemento de menor ou maior valor do vetor. Na prática isso significa que o algoritmo será lento quando o vetor já estiver ordenado, de modo crescente ou decrescente.

A complexidade média do algoritmo é da ordem de $\mathcal{O}(n.log^n)$. Esse valor é obtido quando a escolha do pivô resultar na divisão em dois vetores de tamanhos parecidos, ou seja, o pivô deve ser movido para próximo do centro do vetor ordenado numa dada iteração. Considerando vetores com distribuição aleatória uniforme de seus elementos, essa complexidade é quase sempre atingida.

10.8 Ordenação por Merge-sort

A ordenação por Merge-sort, também chamada por intercalação, é um dos métodos mais rápidos para ordenação, porém apresenta um custo adicional de armazenamento durante sua aplicação. Seu princípio de funcionamento é muito parecido com a ideia do Shell sort, porém trabalhando com a divisão de cada vetor sempre em dois subvetores, até que se tenha vetores de tamanho k suficientemente pequeno ($k = 1$ na formulação original feita por Von Neumann em 1945). Esses vetores são então ordenados por um método qualquer de ordenação e depois são intercalados, dois a dois, até que se chegue ao vetor equivalente ao vetor original. Graficamente o processo é visto na Figura 10.16, considerando um vetor de 7 posições e $k = 1$.

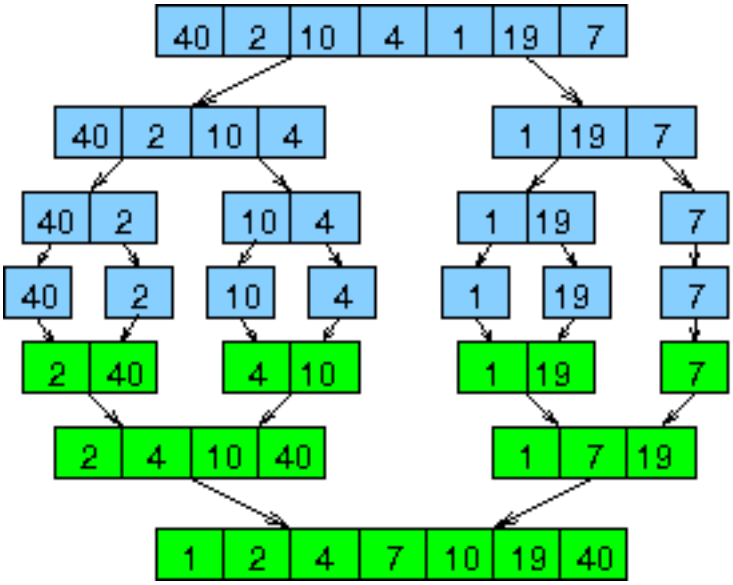

Figura 10.16: Sequência de operações no merge-sort

Nessa figura se observa que o vetor original é sucessivamente particionado até que tenhamos apenas vetores com um elemento. Uma vez obtidos os sete

vetores, se passa a realizar os *merges*, sendo que a cada intercalação se constrói um vetor já ordenado. Esse processo é trivial na primeira etapa em que os dois vetores unidos possuem apenas um elemento. A partir desse ponto é necessário um esquema para juntar dois vetores mantendo a ordenação no vetor resultante, o que é feito de modo também simples.

A intercalação de vetores com dois ou mais elementos é feita "esvaziando" os vetores de entrada, que são copiados para o vetor resultante de forma ordenada. Por exemplo, considerando os vetores { 2, 40 } e { 4, 10 }, primeiro inserimos 2 no vetor resultante, uma vez que ele é o menor elemento entre 2 e 4. Em seguida inserimos 4, pois ele é o menor elemento entre 4 e 40. Depois comparamos 10 e 40, inserindo 10 que é o menor valor, o que conclui a inserção de todos os elementos do segundo vetor. A partir disso se insere os demais elementos do primeiro vetor, que ainda não foram inseridos, resultando no vetor { 2, 4, 10, 40 }.

10.8.1 Projeto e implementação do método

O merge sort pode ser implementado de forma iterativa ou recursiva. Como indicado na seção anterior, o processo é composto por duas fases, uma em que se divide o vetor seguidamente e outra em que se juntam os vetores dois a dois.

Na solução recursiva, apresentada na Figura 10.17, o que se faz é chamar recursivamente a divisão do vetor (linhas 7 e 8) até que se chegue a vetores unitários (na formulação original) ou menores que um dado tamanho, interrompendo a recursão. A partir desse ponto se intercala os vetores criados a cada recursão, até chegar ao vetor original.

Considerando as hipóteses de parada, quando a recursão é interrompida antes de se chegar aos vetores unitários é preciso ordenar seus elementos antes de partir para a fase de merge. Caso a parada ocorra com vetores unitários, então é o processo de merge que faz a ordenação completa, uma vez que um vetor com apenas um elemento está implicitamente ordenado.

10.8.2 Análise de complexidade

O algoritmo pode ser separado em duas partes, que serão repetidas em todas as recursões executadas, sendo que seu tempo de execução será proporcional à multiplicação da complexidade de uma recursão pelo total de recursões. Mostramos agora como cada parte é determinada.

A primeira separa o vetor em partes menores, o que pode ser feito recursivamente em tempo constante, isto é, $\mathcal{O}(1)$, pois envolve apenas o cálculo dos índices dos extremos de cada metade do vetor nas chamadas recursivas.

Na segunda parte do algoritmo se faz a intercalação entre pares de vetores, nos

```
1  void merge_sort(int i, int j, int a[], int aux[])
2  {int k, mid, ptr_esq, ptr_dir;
3
4    if (j <= i)
5       return; // termina recursão se vetor vazio/unitário
6    mid = (i + j) / 2;
7    merge_sort(i, mid, a, aux);      // recursão da parte inferior
8    merge_sort(mid + 1, j, a, aux);  // recursão da parte superior
9
10    ptr_esq = i;  ptr_dir = mid + 1;   k = 0; // início de cada vetor
11    while (ptr_esq <= mid && ptr_dir <= j) // laço do merge
12    { if (a[ptr_esq] < a[ptr_dir]) // menor valor no vetor inferior
13          { aux[k] = a[ptr_esq];   ptr_esq++;  }
14        else
15          { aux[k] = a[ptr_dir];   ptr_dir++;  }
16      k++;
17    }
18    if (ptr_esq > mid)  // se terminou o vetor anterior
19        { while (ptr_dir <= j)
20          { aux[k] = a[ptr_dir];   ptr_dir++;  }
21        }
22      else // ou se terminou o vetor superior
23        { while (ptr_esq <= mid)
24          { aux[k] = a[ptr_esq];   ptr_esq++;  }
25        }
26    for (k = i; k <= j; k++)
27       a[k] = aux[k]; // copia vetor aux no vetor a
28  }
```

Figura 10.17: Implementação do Merge Sort em C.

laços entre as linhas 10 a 25. Esses laços são executados em tempo de ordem da soma do número de elementos dos vetores a serem intercalados, ou seja, em $\mathcal{O}(n)$. Assim, uma execução do algoritmo tomaria $\mathcal{O}(n+1)$, que pode ser aproximado para $\mathcal{O}(n)$.

Como cada uma das partes é executada uma vez para cada chamada recursiva, é preciso determinar o total de recursões que serão realizadas. Como a condição de parada é termos um vetor de uma posição e a cada recursão se divide o vetor em dois, então o número total de recursões é dado por $\log n$. Isso resulta em uma complexidade do método como sendo $\mathcal{O}(n.\log n)$.

10.9 Ordenação por Bucket sort

Diferente dos métodos apresentados até aqui, este método faz a ordenação sem comparar os elementos do vetor entre si. A ideia por trás do algoritmo é classificar os elementos do vetor, separando-os em determinadas categorias de valor. O processo supõe a existência de "baldes" (por isso o nome do algoritmo) em que serão colocados cada elemento do vetor, separando-os por sua categoria. A eficiência do algoritmo depende da quantidade de categorias/baldes e da distribuição uniforme de valores presentes no vetor, uma vez que quanto menos elementos por balde mais rápida será a ordenação final.

O que se faz neste algoritmo é definir faixas de valores para cada balde. Por exemplo, se os valores armazenados forem inteiros positivos entre 0 e 100.000, então podemos ter mil baldes armazenando valores em intervalos de 100 unidades cada, ou seja, num balde vão valores de 0 a 99, noutro vão de 100 a 199 e assim por diante.

Após o processo de separação é natural que alguns baldes contenham mais do que um elemento do vetor. Para esses casos se aplica um algoritmo de ordenação para os valores armazenados no balde. Esse algoritmo pode ser uma reaplicação do *bucket sort* para os valores em cada balde (o que aumenta ainda mais a necessidade de memória) ou algo como o *insertion sort* (pois teoricamente são poucos valores). Depois de ter os baldes ordenados basta reagrupar todos os valores no vetor original.

10.9.1 Projeto e implementação do método

Como indicado em sua descrição básica, o algoritmo começa definindo o número de baldes que serão utilizados. Esse número não pode ser muito pequeno, evitando que cada balde tenha muitos elementos, nem muito grande, evitando que se gaste muita memória. Aqui também é preciso definir qual o tipo de dados usado em cada balde para criar uma pilha dos elementos armazenados no balde. Essa pilha pode ser criada usando uma estrutura estática, em que temos que prever quantos elementos irão para cada balde, ou dinâmica, em que o processo de ordenação no balde é mais lento. Na implementação apresentada aqui usaremos uma estrutura estática, supondo que cada balde conterá até 5 vezes o número de valores do que a média esperada, como visto na linha 3 da Figura 10.18.

Após essas definições é preciso implementar efetivamente o algoritmo descrito no começo desta seção. O primeiro passo é preparar os baldes, marcando o topo de suas pilhas (linhas 10 e 11), com o topo apontando para a posição zero de cada balde.

Em seguida se faz a alocação dos elementos do vetor nos baldes apropriados,

```
1  typedef struct bucket {
2    int topo;
3    int pilha[5*tam/nbaldes]; } Balde;
4
5  void BucketSort(int arr[], int tam) {
6    int i, j, pos;
7    Balde baldes[nbaldes];
8
9  // Inicializa o topo dos baldes
10   for (i = 0; i < nbaldes; ++i)
11     baldes[i].topo = 0;
12
13 // Insere os elementos nos respectivos baldes
14   for (i = 0; i < tam; ++i) {
15     pos = calcIndiceBalde(arr[i]);
16     baldes[pos].pilha[baldes[pos].topo] = arr[i];
17     (baldes[pos].topo)++;
18   }
19
20 // Ordena os elementos nos baldes, fazendo com que o topo da pilha
21 // contenha o menor valor (só para baldes com 2 ou mais elementos)
22   for (i = 0; i < nbaldes; ++i) {
23     if (baldes[i].topo > 1)
24         InsertionSortDecreasing(baldes[i].pilha, baldes[i].topo);
25   }
26
27 // Devolve elementos ao vetor original
28   for (j = 0, i = 0; i < nbaldes; ++i) {
29     int k = baldes[i].topo - 1;
30     while (k >= 0) {
31       arr[j] = baldes[i].pilha[k];
32       k--;   j++;
33     }
34   }
35 }
```

Figura 10.18: Implementação do Bucket sort em C.

usando a estrutura de pilha escolhida. O processo é simples, bastando verificar em qual balde cada elemento deve ir, o que é feito com a função *calcIndiceBalde*, chamada na linha 15 e que não será descrita aqui. Por exemplo, para os baldes anteriormente definidos (de 0 a 99 e assim sucessivamente), o elemento 528 deve

ser colocado no balde contendo elementos entre 500 e 599.

Uma vez terminada a alocação dos elementos do vetor é preciso fazer a ordenação do conteúdo em cada um dos baldes. Como mencionado, isso pode ser feito aplicando novamente o algoritmo, agora com os elementos de cada balde por vez, ou então usando um algoritmo de ordenação simples, como o de ordenação por inserção, que é a solução apresentada na Figura 10.18 (linha 24). Aqui deve ficar claro que o uso de estruturas dinâmicas ou estáticas implica em diferenças na execução da ordenação, sendo mais simples se feita numa estrutura estática.

Devemos observar que em nossa solução optamos por fazer a ordenação de modo que a pilha tenha valores crescentes indo do topo à sua base. Isso facilita o processo de devolução dos valores nos baldes para o vetor original, fazendo o esvaziamento da pilha. Além disso, para os baldes com zero ou um elemento (topo menor ou igual a 1) não fazemos a ordenação, a qual seria desnecessária.

Com todos os baldes ordenados o próximo passo é devolver os elementos para o vetor original. Aqui o processo é bastante simples, pois como os baldes contém elementos ordenados basta copiar sucessivamente esses elementos para o vetor original, o que é feito nas linhas 28 a 34, observando que como o topo do balde aponta para a próxima posição a ser preenchida na pilha, então começamos a copiar da posição imediatamente abaixo do topo.

10.9.2 Análise de complexidade

Na análise do algoritmo é possível observar que todas as operações, exceto o laço em que se faz as chamadas para o algoritmo de ordenação por inserção, ocorrem em $\mathcal{O}(n)$ para um vetor de n posições. Assim, é necessário determinar o custo do laço para ordenação de cada balde para termos o custo total do algoritmo.

Como o algoritmo de ordenação por inserção tem custo de $\mathcal{O}(n^2)$, esse seria o pior caso, quando todos os valores do vetor sejam alocados em um único balde. Entretanto, o algoritmo é planejado para vetores com valores uniformemente distribuídos, o que teoricamente evita que baldes tenham muitos elementos.

Desse modo, supondo que a premissa de distribuição dos valores seja observada, então cada balde terá n/b elementos, em que b é o número de baldes. Supondo ainda que $b = n$, então o custo da ordenação seria unitário e o custo total do algoritmo seria de $\mathcal{O}(n)$. Mesmo sabendo que normalmente não teremos essa situação ideal, o algoritmo é de fato bastante rápido, embora tenha um custo de armazenamento em memória bastante alto.

Capítulo 11

Métodos básicos de busca

11.1 Métodos de busca em listas ordenadas

O processo de busca por um elemento em uma lista ordenada precisa ser eficiente. Isso ocorre pois em geral se usa listas para implementar o armazenamento de grandes quantidades de dados, que são acessados por informações específicas. Dependendo da estrutura de armazenamento utilizada podem ser implementados diferentes métodos de busca, como a sequencial ou a binária. Na sequência esses métodos são examinados.

11.1.1 Busca sequencial

Na busca sequencial, como o nome já indica, a busca por um dado elemento começa em uma das extremidades da lista e continua, de posição em posição, até que ele seja encontrado. Isso significa, na prática, que se começamos do primeiro elemento (L_1), então o próximo elemento a ser verificado é aquele que o sucede na lista (L_2), seguido pela verificação de L_3 e assim por diante, até que o elemento L_i corresponda ao elemento procurado ou se chegue ao final da lista.

Esse é o método de busca que temos utilizado nos capítulos anteriores. Vale lembrar que se a lista ou vetor estiver ordenado, então é possível terminar a busca assim que o elemento buscado for maior do que o elemento verificado no vetor. A busca sequencial é bastante simples, sendo também o método que funciona quando se implementa listas com estruturas dinâmicas. A Figura 11.1 contém o código genérico de uma função de busca sequencial, considerando estruturas dinâmicas.

O laço nas linhas 7 e 8 representa efetivamente o processo de busca, considerando que a função NOT_EQUAL verifica se o elemento buscado corresponde ao elemento atual na lista. O teste na linha 10 verifica se a busca terminou com sucesso ou não. O tratamento dado nas linhas seguintes depende do que se pre-

```
1   struct lista *buscaseq(struct lista *head, tipo_chave elemento)
2   // função NOT_EQUAL retorna 0 se aux contém o elemento procurado
3   {int i, pos, antes;
4    struct lista *aux;
5
6     aux = head;
7     while (aux->prox && NOT_EQUAL(elemento, aux->chave) )
8         aux = aux->prox;  // função NOT_EQUAL retorna 0 se aux
9                           // contém o elemento procurado
10    if (NOT_EQUAL(elemento,aux->chave) )
11        {// Não encontrou o elemento buscado.
12         // Fazer o tratamento dessa ocorrência!
13        }
14      else
15        {// aux contém o endereço do elemento buscado
16          return (aux);
17        }
18  }
```

Figura 11.1: Busca sequencial por um elemento em lista dinâmica.

tende fazer com a busca, sendo que tipicamente se retorna o endereço da posição que contém o elemento buscado no caso de sucesso na busca.

11.1.2 Busca binária

O problema com a busca sequencial, quando aplicada a vetores ordenados, é que ela desconsidera a propriedade essencial de vetores ordenados, que é a de que $\forall i, j \mid i < j \Rightarrow A_i \leq A_j$. Isso, na prática, faz com que o processo de busca sequencial seja ineficiente. Por exemplo, se uma lista contiver todos os valores inteiros entre 1 e 1000 e buscarmos por 720, na busca sequencial testaremos todos os valores menores que 720 até chegar a ele.

Se usarmos listas ligadas esse é, infelizmente, o procedimento possível, a menos que se use processos de indexação razoavelmente elaborados e que demandam maior ocupação de espaço na memória. Entretanto, se o armazenamento dos elementos for feito de modo estritamente sequencial em um vetor ordenado (e não na forma de lista ligada), torna-se possível usar uma estratégia conhecida como busca binária.

Na busca binária o que se faz é considerar que se o elemento buscado não estiver em uma das extremidades do vetor, então devemos avaliar o elemento que está no ponto médio entre as extremidades. Assim, no exemplo do vetor

preenchido por valores correspondentes aos índices, dado no início desta seção, se avaliaria o elemento da posição 500 (que tem valor 500 pela definição do vetor). A sequência do processo de busca é guiada, então, pela comparação entre o elemento procurado e o elemento da posição avaliada, desprezando-se a "metade" do vetor que não conterá o elemento procurado, de acordo com a propriedade de ordenação.

Isso é possível pois como a lista está armazenada na forma de vetor ordenado, é obviamente inútil procurar o valor 720 entre os elementos que contém valores menores ou iguais a 500. Assim, temos um "novo" vetor, que começa com valor 500 e termina com o valor 1000. O elemento intermediário entre esses dois estará na posição 750 (valor 750). Mais uma vez, como o vetor está ordenado, é inútil procurar o elemento nas posições acima de 750, o que nos leva a um novo vetor, que começa com 500 e termina com 750.

O processo se repete, ora descartando a parte inferior do vetor, ora descartando a parte superior. Isso é ilustrado na Figura 11.2, em que se pode ver todas as mudanças no elemento verificado a cada passo. Nessa figura as regiões sombreadas identificam as partes do vetor que são desconsideradas a cada iteração.

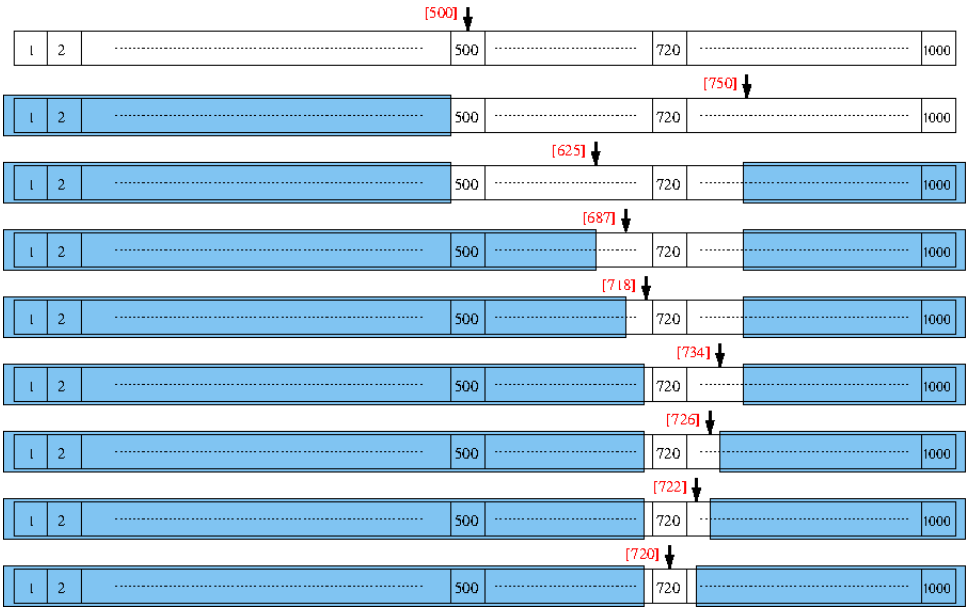

Figura 11.2: Sequência de elementos buscados no vetor em caso de busca binária.

Com isso, em vez de testarmos 720 elementos (da posição 1 até a posição 720), testamos apenas 11 elementos (contando as extremidades iniciais do vetor). Na verdade, a quantidade de posições avaliadas no vetor é limitada em $(k + 2)$ se o número de elementos no vetor for menor ou igual a 2^k.

O código para a busca binária é visto na Figura 11.3. Nela se observa que

inicialmente se verifica as duas extremidades do vetor. Se o elemento procurado não estiver nelas se passa ao laço `while` das linhas 12 a 18, sempre atualizando as extremidades da busca. O resultado, quando a busca é bem sucedida, é retornado pelo valor de mid.

```
1   int buscabin(tipo *vetor, tipo_chave elemento, int N)
2   // função NOT_EQUAL retorna 0 se aux contém o elemento procurado
3   // função IS_GREATER retorna 1 se o elemento procurado for maior
4   // que o elemento na posição mid
5   {int min=0, max=N, mid;
6
7     if (! NOT_EQUAL(elemento, vetor[min]))
8         return(vetor[min]);
9     if (! NOT_EQUAL(elemento, vetor[max]))
10         return(vetor[max]);
11    mid = (min + max) / 2;
12    while (NOT_EQUAL(elemento, vetor[mid]) && min < max)
13    { if (IS_GREATER(elemento, vetor[mid]))
14         min = mid; // elemento está na parte superior do vetor
15       else
16         max = mid; // elemento está na parte inferior do vetor
17     mid = (min + max) / 2;
18    }
19    if (NOT_EQUAL(elemento, vetor[mid]) )
20        { // Não encontrou o elemento. Tratar essa ocorrência!
21        }
22      else
23        { // O elemento buscado está na posição mid, que é retornada
24          return (mid);
25        }
26  }
```

Figura 11.3: Busca binária por um elemento em vetor ordenado.

11.1.3 Busca por interpolação

A busca binária é bastante eficiente, porém deixa de aproveitar uma condição que aparece em vetores que tenham distribuição uniforme de valores. Esse é o caso do vetor usado como exemplo, apesar de sabermos que esse não é um caso muito realístico. Entretanto, muitos vetores apresentam distribuições uniformes, ou pelo menos não totalmente aleatórias, nos valores armazenados.

Assim, é possível interpolar o valor procurado pelos índices do vetor, procurando fazer a busca orientada por sua possível posição. A interpolação consiste em determinar a provável localização do elemento buscado a partir de uma relação entre o número de posições do vetor, dos seus valores mínimo e máximo e do valor buscado. A equação 11.1 mostra como o índice da posição a ser comparada deve ser calculado:

$$Posição = (I_{max} - I_{min}) \times \frac{Valor}{V[I_{max}] - V[I_{min}]} \tag{11.1}$$

Nessa equação temos que I_{min} e I_{max} representam os índices da primeira e da última posição do vetor de busca e $V[I_{min}]$ e $V[I_{max}]$ são os valores respectivamente armazenados nessas posições. Com isso, a posição a ser comparada é aquela que tem mais probabilidade de estar próxima do valor buscado.

O código apresentado na Figura 11.4 trás a implementação da busca por interpolação. Como se pode observar, a única modificação em relação à busca binária envolve o cálculo do ponto intermediário no vetor. No caso da busca binária se usa a posição média entre as duas extremidades, enquanto no caso da busca por interpolação se usa a fórmula de interpolação da Equação 11.1. Todo o restante do algoritmo é estritamente idêntico aos dois algoritmos de busca.

Embora a busca por interpolação seja teoricamente mais rápida que a busca binária, a análise do número máximo de comparações é equivalente ao visto na seção anterior. Isso significa que na prática o número de comparações também fica limitado em $(k + 2)$ se o número de elementos no vetor for menor ou igual a 2^k, que é um valor relativamente pequeno, pois para um vetor de um milhão de posições faríamos no máximo 22 comparações. Como o número de operações para determinar o próximo elemento é significativamente maior (duas subtrações, uma divisão e uma multiplicação, além de dois acessos ao vetor, contra uma soma e uma divisão), então a busca por interpolação é aplicável apenas para vetores muito grandes e que tenham uma distribuição uniforme de valores ao longo do vetor.

```
1   int buscainterpolada(tipo *vetor, tipo_chave elemento, int N)
2   // função NOT_EQUAL retorna 0 se aux contém o elemento procurado
3   // função IS_GREATER retorna 1 se o elemento procurado for maior
4   // que o elemento na posição pos
5   {int min=0, max=N, pos;
6
7     if (! NOT_EQUAL(elemento, vetor[min]))
8         return(vetor[min]);
9     if (! NOT_EQUAL(elemento, vetor[max]))
10        return(vetor[max]);
11    pos = (max - min) * elemento / (vetor[max] - vetor[min]);
12    while (NOT_EQUAL(elemento, vetor[pos]) && min < max)
13    { if (IS_GREATER(elemento, vetor[pos]))
14         min = pos; // elemento está na parte superior do vetor
15       else
16         max = pos; // elemento está na parte inferior do vetor
17     pos = (max - min) * elemento / (vetor[max] - vetor[min]);
18    }
19    if (NOT_EQUAL(elemento, vetor[pos]) )
20        { // Não encontrou o elemento. Tratar essa ocorrência!
21        }
22      else
23        { // O elemento buscado está na posição pos, que é retornada
24          return (mid);
25        }
26  }
```

Figura 11.4: Busca interpolada por um elemento em vetor ordenado.