

Apostila de Estruturas de Dados

**Profs. Waldemar Celes e José Lucas Rangel
PUC-RIO - Curso de Engenharia - 2002**

Apresentação

A disciplina de *Estruturas de Dados* (ED) está sendo ministrada em sua nova versão desde o segundo semestre de 1998. Trata-se da segunda disciplina de informática oferecida no curso de Engenharia da PUC-Rio. Na primeira disciplina, *Introdução à Ciência da Computação* (ICC), são apresentados os conceitos fundamentais de programação. ICC, em sua versão mais nova, utiliza a linguagem *Scheme*, de fácil aprendizado, o que permite a discussão de diversos conceitos de programação num curso introdutório. Isso acontece porque Scheme, como a linguagem LISP da qual descende, é uma linguagem funcional, baseada em conceitos familiares aos alunos, como a definição de funções e sua aplicação em expressões que devem ser avaliadas.

O enfoque do curso de Estruturas de Dados é diferente. Discutem-se técnicas de programação e estruturação de dados para o desenvolvimento de programas eficientes. Adota-se a linguagem de programação C. Apesar de reconhecermos as dificuldades na aprendizagem da linguagem C, optamos por sua utilização neste curso simplesmente porque C é a linguagem básica da programação do UNIX, da Internet, do Windows, do Linux. Além de C, usam-se nestes sistemas e em aplicações desenvolvidas para eles linguagens derivadas de C, como C++ e Java. Um ponto adicional a favor da escolha de C é que o estudo de várias disciplinas posteriores a ED será facilitado se os alunos já puderem programar com desenvoltura nessa linguagem.

Este curso foi idealizado e montado pelo Prof. José Lucas Rangel. Neste semestre, estamos reformulando alguns tópicos, criando outros e alterando a ordem de apresentação. Esta apostila foi reescrita tendo como base a apostila do Prof. Rangel, utilizada nos semestres anteriores.

O curso está dividido em três partes. A Parte I apresenta os conceitos fundamentais da linguagem C e discute formas simples de estruturação de dados; a Parte II discute as estruturas de listas e árvores, e suas aplicações; e a Parte III discute algoritmos e estruturas de dados para ordenação e busca.

A apostila apresenta todos os tópicos que serão discutidos em sala de aula, mas recomendamos fortemente que outras fontes (livros, notas de aula, etc.) sejam consultadas.

Rio de Janeiro, 19 de fevereiro de 2002
Waldemar Celes

Índice

1. Conceitos fundamentais	1-1
1.1. Introdução	1-1
1.2. Modelo de um computador	1-1
1.3. Interpretação versus Compilação	1-3
1.4. Exemplo de código em C	1-4
1.5. Compilação de programas em C.....	1-6
1.6. Ciclo de desenvolvimento	1-8
2. Expressões	2-1
2.1. Variáveis.....	2-1
2.2. Operadores	2-4
2.3. Entrada e saída básicas.....	2-8
3. Controle de fluxo.....	3-1
3.1. Decisões com if.....	3-1
3.2. Construções com laços.....	3-4
3.3. Seleção	3-8
4. Funções	4-1
4.1. Definição de funções.....	4-1
4.2. Pilha de execução.....	4-3
4.3. Ponteiro de variáveis.....	4-6
4.4. Recursividade.....	4-10
4.5. Variáveis estáticas dentro de funções	4-11
4.6. Pré-processador e macros	4-12
5. Vetores e alocação dinâmica	5-1
5.1. Vetores	5-1
5.2. Alocação dinâmica	5-3
6. Cadeia de caracteres	6-1
6.1. Caracteres.....	6-1
6.2. Cadeia de caracteres (strings)	6-3
6.3. Vetor de cadeia de caracteres	6-11
7. Tipos estruturados	7-1
7.1. O tipo estrutura.....	7-1
7.2. Definição de "novos" tipos.....	7-4
7.3. Vetores de estruturas	7-6
7.4. Vetores de ponteiros para estruturas.....	7-7
7.5. Tipo união.....	7-9
7.6. Tipo enumeração	7-10
8. Matrizes	8-1
8.1. Alocação estática versus dinâmica	8-1
8.2. Vetores bidimensionais – Matrizes.....	8-2
8.3. Matrizes dinâmicas.....	8-4
8.4. Representação de matrizes	8-6
8.5. Representação de matrizes simétricas	8-9
9. Tipos Abstratos de Dados	9-1

9.1. Módulos e Compilação em Separado	9-1
9.2. Tipo Abstrato de Dados.....	9-3
10. Listas Encadeadas	10-1
10.1. Lista encadeada	10-2
10.2. Implementações recursivas	10-9
10.3. Listas genéricas	10-10
10.4. Listas circulares.....	10-15
10.5. Listas duplamente encadeadas.....	10-16
11. Pilhas.....	10-1
11.1. Interface do tipo pilha	10-2
11.2. Implementação de pilha com vetor	10-2
11.3. Implementação de pilha com lista	10-3
11.4. Exemplo de uso: calculadora pós-fixada.....	10-5
12. Filas	11-1
12.1. Interface do tipo fila	11-1
12.2. Implementação de fila com vetor	11-2
12.3. Implementação de fila com lista	11-5
12.4. Fila dupla	11-7
12.5. Implementação de fila dupla com lista	11-8
13. Árvores	12-1
13.1. Árvores binárias	12-2
13.2. Árvores genéricas	12-9
14. Arquivos	13-1
14.1. Funções para abrir e fechar arquivos	13-2
14.2. Arquivos em modo texto.....	13-3
14.3. Estruturação de dados em arquivos textos.....	13-4
14.4. Arquivos em modo binário.....	13-11
15. Ordenação	14-1
15.1. Ordenação bolha.....	14-1
15.2. Ordenação Rápida	14-9
16. Busca	16-1
16.1. Busca em Vetor.....	16-1
16.2. Árvore binária de busca	16-7
17. Tabelas de dispersão.....	17-1
17.1. Idéia central.....	17-2
17.2. Função de dispersão.....	17-3
17.3. Tratamento de colisão.....	17-4
17.4. Exemplo: Número de Ocorrências de Palavras	17-8
17.5. Uso de callbacks	17-13

1. Conceitos fundamentais

W. Celes e J. L. Rangel

1.1. Introdução

O curso de *Estruturas de Dados* discute diversas técnicas de programação, apresentando as estruturas de dados básicas utilizadas no desenvolvimento de *software*. O curso também introduz os conceitos básicos da linguagem de programação C, que é utilizada para a implementação das estruturas de dados apresentadas. A linguagem de programação C tem sido amplamente utilizada na elaboração de programas e sistemas nas diversas áreas em que a informática atua, e seu aprendizado tornou-se indispensável tanto para programadores profissionais como para programadores que atuam na área de pesquisa.

O conhecimento de linguagens de programação por si só não capacita programadores – é necessário saber usá-las de maneira eficiente. O projeto de um programa engloba a fase de identificação das propriedades dos dados e características funcionais. Uma representação adequada dos dados, tendo em vista as funcionalidades que devem ser atendidas, constitui uma etapa fundamental para a obtenção de programas eficientes e confiáveis.

A linguagem C, assim como as linguagens Fortran e Pascal, são ditas linguagens “convencionais”, projetadas a partir dos elementos fundamentais da arquitetura de von Neuman, que serve como base para praticamente todos os computadores em uso. Para programar em uma linguagem convencional, precisamos de alguma maneira especificar as áreas de memória em que os dados com que queremos trabalhar estão armazenados e, freqüentemente, considerar os endereços de memória em que os dados se situam, o que faz com que o processo de programação envolva detalhes adicionais, que podem ser ignorados quando se programa em uma linguagem como Scheme. Em compensação, temos um maior controle da máquina quando utilizamos uma linguagem convencional, e podemos fazer programas melhores, ou seja, menores e mais rápidos.

A linguagem C provê as construções fundamentais de fluxo de controle necessárias para programas bem estruturados: agrupamentos de comandos; tomadas de decisão (*if-else*); laços com testes de encerramento no início (*while*, *for*) ou no fim (*do-while*); e seleção de um dentre um conjunto de possíveis casos (*switch*). C oferece ainda acesso a apontadores e a habilidade de fazer aritmética com endereços. Por outro lado, a linguagem C não provê operações para manipular diretamente objetos compostos, tais como cadeias de caracteres, nem facilidades de entrada e saída: não há comandos READ e WRITE. Todos esses mecanismos devem ser fornecidos por funções explicitamente chamadas. Embora a falta de algumas dessas facilidades possa parecer uma deficiência grave (deve-se, por exemplo, chamar uma função para comparar duas cadeias de caracteres), a manutenção da linguagem em termos modestos tem trazido benefícios reais. C é uma linguagem relativamente pequena e, no entanto, tornou-se altamente poderosa e eficiente.

1.2. Modelo de um computador

Existem diversos tipos de computadores. Embora não seja nosso objetivo estudar *hardware*, identificamos, nesta seção, os elementos essenciais de um computador. O

conhecimento da existência destes elementos nos ajudará a compreender como um programa de computador funciona.

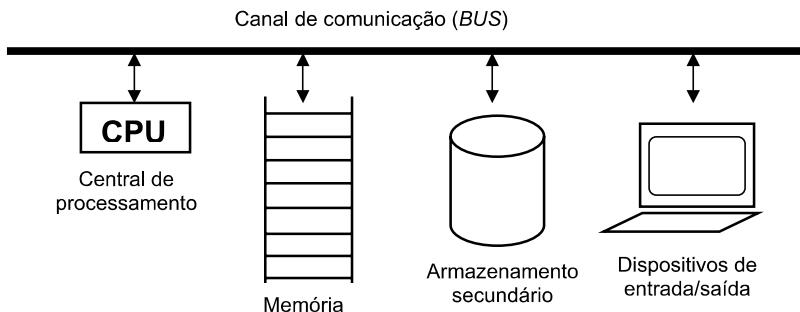


Figura 1.1: Elementos básicos de um computador típico.

A Figura 1.1 identifica os elementos básicos de um computador típico. O canal de comunicação (conhecido como *BUS*) representa o meio para a transferência de dados entre os diversos componentes. Na memória principal são armazenados os programas e os dados no computador. Ela tem acesso randômico, o que significa que podemos endereçar (isto é, acessar) diretamente qualquer posição da memória. Esta memória não é permanente e, para um programa, os dados são armazenados enquanto o programa está sendo executado. Em geral, após o término do programa, a área ocupada na memória fica disponível para ser usada por outras aplicações. A área de armazenamento secundário é, em geral, representada por um disco (disco rígido, disquete, etc.). Esta memória secundária tem a vantagem de ser permanente. Os dados armazenados em disco permanecem válidos após o término dos programas. Esta memória tem um custo mais baixo do que a memória principal, porém o acesso aos dados é bem mais lento. Por fim, encontram-se os dispositivos de entrada e saída. Os dispositivos de entrada (por exemplo, teclado, *mouse*) permitem passarmos dados para um programa, enquanto os dispositivos de saída permitem que um programa exporte seus resultados, por exemplo em forma textual ou gráfica usando monitores ou impressoras.

Armazenamento de dados e programas na memória

A memória do computador é dividida em unidades de armazenamento chamadas *bytes*. Cada byte é composto por 8 *bits*, que podem armazenar os valores zero ou um. Nada além de zeros e uns pode ser armazenado na memória do computador. Por esta razão, todas as informações (programas, textos, imagens, etc.) são armazenadas usando uma codificação numérica na forma binária. Na representação binária, os números são representados por uma seqüência de zeros e uns (no nosso dia a dia, usamos a representação decimal, uma vez que trabalhamos com 10 algarismos). Por exemplo, o número decimal 5 é representado por 101, pois $1*2^2 + 0*2^1 + 1*2^0$ é igual a 5 (da mesma forma que, na base decimal, $456=4*10^2 + 5*10^1 + 6*10^0$). Cada posição da memória (byte) tem um endereço único. Não é possível endereçar diretamente um bit.

Se só podemos armazenar números na memória do computador, como fazemos para armazenar um texto (um documento ou uma mensagem)? Para ser possível armazenar uma seqüência de caracteres, que representa o texto, atribui-se a cada caractere um código

numérico (por exemplo, pode-se associar ao caractere 'A' o código 65, ao caractere 'B' o código 66, e assim por diante). Se todos os caracteres tiverem códigos associados (inclusive os caracteres de pontuação e de formatação), podemos armazenar um texto na memória do computador como uma seqüência de códigos numéricos.

Um computador só pode executar programas em linguagens de máquina. Cada programa executável é uma seqüência de instruções que o processador central interpreta, executando as operações correspondentes. Esta seqüência de instruções também é representada como uma seqüência de códigos numéricos. Os programas ficam armazenados em disco e, para serem executados pelo computador, devem ser carregados (transferidos) para a memória principal. Uma vez na memória, o computador executa a seqüência de operações correspondente.

1.3. *Interpretação versus Compilação*

Uma diferença importante entre as linguagens C e Scheme é que, via de regra, elas são implementadas de forma bastante diferente. Normalmente, Scheme é interpretada e C é compilada. Para entender a diferença entre essas duas formas de implementação, é necessário lembrar que os computadores só executam realmente programas em sua linguagem de máquina, que é específica para cada modelo (ou família de modelos) de computador. Ou seja, em qualquer computador, programas em C ou em Scheme não podem ser executados em sua forma original; apenas programas na linguagem de máquina (λ qual vamos nos referir como M) podem ser efetivamente executados.

No caso da interpretação de Scheme, um programa interpretador (I_M), escrito em M, lê o programa P_S escrito em Scheme e simula cada uma de suas instruções, modificando os dados do programa da forma apropriada. No caso da compilação da linguagem C, um programa compilador (C_M), escrito em M, lê o programa P_C , escrito em C, e traduz cada uma de suas instruções para M, escrevendo um programa P_M cujo efeito é o desejado. Como consequência deste processo, P_M , por ser um programa escrito em M, pode ser executado em qualquer máquina com a mesma linguagem de máquina M, mesmo que esta máquina não possua um compilador.

Na prática, o programa fonte e o programa objeto são armazenados em arquivos em disco, aos quais nos referimos como arquivo fonte e arquivo objeto. As duas figuras a seguir esquematizam as duas formas básicas de implementação de linguagens de programação.

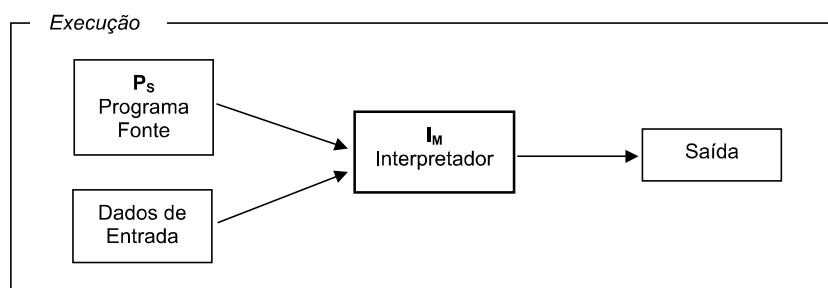


Figura 1.2: Execução de programas com linguagem interpretada.

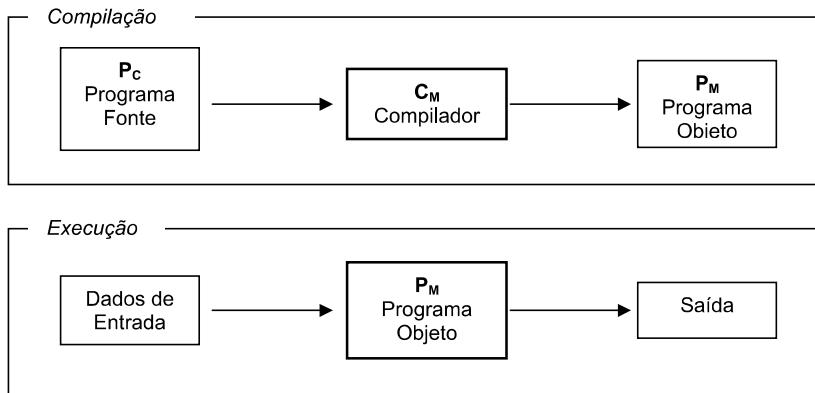


Figura 1.3: Execução de programas com linguagem compilada.

Devemos notar que, na Figura 1.2, o programa fonte é um dado de entrada a mais para o interpretador. No caso da compilação, Figura 1.3, identificamos duas fases: na primeira, o programa objeto é a saída do programa compilador e, na segunda, o programa objeto é executado, recebendo os dados de entrada e gerando a saída correspondente.

Observamos que, embora seja comum termos linguagens funcionais implementadas por interpretação e linguagens convencionais por compilação, há exceções, não existindo nenhum impedimento conceitual para implementar qualquer linguagem por qualquer dos dois métodos, ou até por uma combinação de ambos. O termo “máquina” usado acima é intencionalmente vago. Por exemplo, computadores idênticos com sistemas operacionais diferentes devem ser considerados “máquinas”, ou “plataformas”, diferentes. Assim, um programa em C, que foi compilado em um PC com Windows, não deverá ser executado em um PC com Linux, e vice-versa.

1.4. Exemplo de código em C

Para exemplificar códigos escritos em C, consideremos um programa que tem a finalidade de converter valores de temperatura dados em Celsius para Fahrenheit. Este programa define uma função principal que capture um valor de temperatura em Celsius, fornecido via teclado pelo usuário, e exibe como saída a temperatura correspondente em Fahrenheit. Para fazer a conversão, é utilizada uma função auxiliar. O código C deste programa exemplo é mostrado abaixo.

```

/* Programa para conversão de temperatura */

#include <stdio.h>

float converte (float c)
{
    float f;
    f = 1.8*c + 32;
    return f;
}

```

```

int main (void)
{
    float t1;
    float t2;

    /* mostra mensagem para usuario */
    printf("Digite a temperatura em Celsius: ");

    /* captura valor entrado via teclado */
    scanf("%f",&t1);

    /* faz a conversao */
    t2 = converte(t1);

    /* exibe resultado */
    printf("A temperatura em Fahrenheit é: %f\n", t2);

    return 0;
}

```

Um programa em C, em geral, é constituído de diversas pequenas funções, que são independentes entre si – não podemos, por exemplo, definir uma função dentro de outra. Dois tipos de ambientes são caracterizados em um código C. O ambiente global, externo às funções, e os ambientes locais, definidos pelas diversas funções (lembrando que os ambientes locais são independentes entre si). Podem-se inserir comentários no código fonte, iniciados com `/*` e finalizados com `*/`, conforme ilustrado acima. Devemos notar também que comandos e declarações em C são terminados pelo caractere ponto-e-vírgula `(;)`.

Um programa em C tem que, obrigatoriamente, conter a função principal (`main`). A execução de um programa começa pela função principal (a função `main` é automaticamente chamada quando o programa é carregado para a memória). As funções auxiliares são chamadas, direta ou indiretamente, a partir da função principal.

Em C, como nas demais linguagens “convencionais”, devemos reservar área de memória para armazenar cada dado. Isto é feito através da declaração de variáveis, na qual informamos o tipo do dado que iremos armazenar naquela posição de memória. Assim, a declaração `float t1;`, do código mostrado, reserva um espaço de memória para armazenarmos um valor real (ponto flutuante – `float`). Este espaço de memória é referenciado através do símbolo `t1`.

Uma característica fundamental da linguagem C diz respeito ao tempo de vida e à visibilidade das variáveis. Uma variável (local) declarada dentro de uma função “vive” enquanto esta função está sendo executada, e nenhuma outra função tem acesso direto a esta variável. Outra característica das variáveis locais é que devem sempre ser explicitamente inicializadas antes de seu uso, caso contrário conterão “lixo”, isto é, valores indefinidos.

Como alternativa, é possível definir variáveis que sejam externas às funções, isto é, variáveis globais, que podem ser acessadas pelo nome por qualquer função subsequente

(são “visíveis” em todas as funções que se seguem à sua definição). Além do mais, devido às variáveis externas (ou globais) existirem permanentemente (pelo menos enquanto o programa estiver sendo executado), elas retêm seus valores mesmo quando as funções que as acessam deixam de existir. Embora seja possível definir variáveis globais em qualquer parte do ambiente global (entre quaisquer funções), é prática comum defini-las no início do arquivo-fonte.

Como regra geral, por razões de clareza e estruturação adequada do código, devemos evitar o uso indisciplinado de variáveis globais e resolver os problemas fazendo uso de variáveis locais sempre que possível. No próximo capítulo, discutiremos variáveis com mais detalhe.

1.5. Compilação de programas em C

Para desenvolvermos programas em uma linguagem como C, precisamos de, no mínimo, um editor e um compilador. Estes programas têm finalidades bem definidas: com o editor de textos, escrevemos os programas fontes, que são salvos em arquivos¹; com o compilador, transformamos os programas fontes em programas objetos, em linguagem de máquina, para poderem ser executados. Os programas fontes são, em geral, armazenados em arquivos cujo nome tem a extensão “.c”. Os programas executáveis possuem extensões que variam com o sistema operacional: no Windows, têm extensão “.exe”; no Unix (Linux), em geral, não têm extensão.

Para exemplificar o ciclo de desenvolvimento de um programa simples, consideremos que o código apresentado na seção anterior tenha sido salvo num arquivo com o nome `prog.c`. Devemos então compilar o programa para gerarmos um executável. Para ilustrar este processo, usaremos o compilador `gcc`. Na linha de comando do sistema operacional, fazemos:

```
> gcc -o prog prog.c
```

Se não houver erro de compilação no nosso código, este comando gera o executável com o nome `prog` (`prog.exe`, no Windows). Podemos então executar o programa:

```
> prog
Digite a temperatura em Celsius: 10
A temperatura em Fahrenheit vale: 50.000000
>
```

Em itálico, representamos as mensagens do programa e, em negrito, exemplificamos um dado fornecido pelo usuário via teclado.

Programas com vários arquivos fontes

Os programas reais são, naturalmente, maiores. Nestes casos, subdividimos o fonte do programa em vários arquivos. Para exemplificar a criação de um programa com dois arquivos, vamos considerar que o programa para conversão de unidades de temperatura

¹ Podemos utilizar qualquer editor de texto para escrever os programas fontes, exceto editores que incluem caracteres de formatação (como o Word do Windows, por exemplo).

apresentado anteriormente seja dividido em dois fontes: o arquivo `converte.c` e o arquivo `principal.c`. Teríamos que criar dois arquivos, como ilustrado abaixo:

Arquivo `converte.c`:

```
/* Implementação do módulo de conversão */

float converte (float c)
{
    float f;
    f = 1.8*c + 32;
    return f;
}
```

Arquivo `principal.c`:

```
/* Programa para conversão de temperatura */

#include <stdio.h>

float converte (float c);

int main (void)
{
    float t1;
    float t2;

    /* mostra mensagem para usuário */
    printf("Entre com temperatura em Celsius: ");

    /* captura valor entrado via teclado */
    scanf("%f",&t1);

    /* faz a conversão */
    t2 = converte(t1);

    /* exibe resultado */
    printf("A temperatura em Fahrenheit vale: %f\n", t2);

    return 0;
}
```

Embora o entendimento completo desta organização de código não fique claro agora, interessa-nos apenas mostrar como geramos um executável de um programa com vários arquivos fontes. Uma alternativa é compilar tudo junto e gerar o executável como anteriormente:

```
> gcc -o prog converte.c principal.c
```

No entanto, esta não é a melhor estratégia, pois se alterarmos a implementação de um determinado módulo não precisaríamos re-compilar os outros. Uma forma mais eficiente é compilarmos os módulos separadamente e depois ligar os diversos módulos gerados para criar um executável.

```
> gcc -c converte.c
> gcc -c principal.c
> gcc -o prog converte.o principal.o
```

A opção `-c` do compilador `gcc` indica que não queremos criar um executável, apenas gerar o arquivo objeto (com extensão “`.o`” ou “`.obj`”). Depois, invocamos `gcc` para fazer a ligação dos objetos, gerando o executável.

1.6. Ciclo de desenvolvimento

Programas como editores, compiladores e ligadores são às vezes chamados de “ferramentas”, usadas na “Engenharia” de Software. Exceto no caso de programas muito pequenos (como é o caso de nosso exemplo), é raro que um programa seja composto de um único arquivo fonte. Normalmente, para facilitar o projeto, os programas são divididos em vários arquivos. Como vimos, cada um desses arquivos pode ser compilado em separado, mas para sua execução é necessário reunir os códigos de todos eles, sem esquecer das bibliotecas necessárias, e esta é a função do *ligador*.

A tarefa das bibliotecas é permitir que funções de interesse geral estejam disponíveis com facilidade. Nossa exemplo usa a biblioteca de entrada/saída padrão do C, `stdio`, que oferece funções que permitem a captura de dados a partir do teclado e a saída de dados para a tela. Além de bibliotecas preparadas pelo fornecedor do compilador, ou por outros fornecedores de *software*, podemos ter bibliotecas preparadas por um usuário qualquer, que pode “empacotar” funções com utilidades relacionadas em uma biblioteca e, dessa maneira, facilitar seu uso em outros programas.

Em alguns casos, a função do ligador é executada pelo próprio compilador. Por exemplo, quando compilamos o primeiro programa `prog.c`, o ligador foi chamado automaticamente para reunir o código do programa aos códigos de `scanf`, `printf` e de outras funções necessárias à execução independente do programa.

Verificação e Validação

Outro ponto que deve ser observado é que os programas podem conter (e, em geral, contêm) erros, que precisam ser identificados e corrigidos. Quase sempre a verificação é realizada por meio de testes, executando o programa a ser testado com diferentes valores de entrada. Identificado um ou mais erros, o código fonte é corrigido e deve ser novamente verificado. O processo de compilação, ligação e teste se repete até que os resultados dos testes sejam satisfatórios e o programa seja considerado validado. Podemos descrever o ciclo através da Figura 1.4.

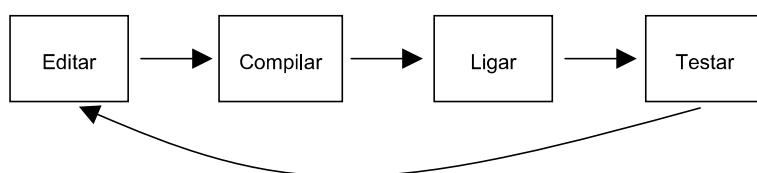


Figura 1.4: Ciclo de desenvolvimento.

Este ciclo pode ser realizado usando programas (editor, compilador, ligador) separados ou empregando um “ambiente integrado de desenvolvimento” (*integrated development environment*, ou IDE). IDE é um programa que oferece janelas para a edição de programas e facilidades para abrir, fechar e salvar arquivos e para compilar, ligar e executar programas. Se um IDE estiver disponível, é possível criar e testar um programa, tudo em um mesmo ambiente, e todo o ciclo mencionado acima acontece de maneira mais confortável dentro de um mesmo ambiente, de preferência com uma interface amigável.

2. Expressões

W. Celes e J. L. Rangel

Em C, uma expressão é uma combinação de variáveis, constantes e operadores que pode ser avaliada computacionalmente, resultando em um valor. O valor resultante é chamado de *valor da expressão*.

2.1. Variáveis

Podemos dizer que uma variável representa um espaço na memória do computador para armazenar determinado tipo de dado. Na linguagem C, todas as variáveis devem ser explicitamente declaradas. Na declaração de uma variável, obrigatoriamente, devem ser especificados seu *tipo* e seu *nome*: o nome da variável serve de referência ao dado armazenado no espaço de memória da variável e o tipo da variável determina a natureza do dado que será armazenado.

Tipos básicos

A linguagem C oferece alguns tipos básicos. Para armazenar valores inteiros, existem três tipos básicos: `char`, `short int`, `long int`. Estes tipos diferem entre si pelo espaço de memória que ocupam e consequentemente pelo intervalo de valores que podem representar. O tipo `char`, por exemplo, ocupa 1 byte de memória (8 bits), podendo representar 2^8 ($=256$) valores distintos. Todos estes tipos podem ainda ser modificados para representarem apenas valores positivos, o que é feito precedendo o tipo com o modificador “sem sinal” – `unsigned`. A tabela abaixo compara os tipos para valores inteiros e suas representatividades.

Tipo	Tamanho	Representatividade
<code>char</code>	1 byte	-128 a 127
<code>unsigned char</code>	1 byte	0 a 255
<code>short int</code>	2 bytes	-32 768 a 32 767
<code>unsigned short int</code>	2 bytes	0 a 65 535
<code>long int</code>	4 bytes	-2 147 483 648 a 2 147 483 647
<code>unsigned long int</code>	4 bytes	4 294 967 295

Os tipos `short int` e `long int` podem ser referenciados simplesmente com `short` e `long`, respectivamente. O tipo `int` puro é mapeado para o tipo inteiro natural da máquina, que pode ser `short` ou `long`. A maioria das máquinas que usamos hoje funcionam com processadores de 32 bits e o tipo `int` é mapeado para o inteiro de 4 bytes (`long`).¹

O tipo `char` é geralmente usado apenas para representar códigos de caracteres, como veremos nos capítulos subsequentes.

¹ Um contra-exemplo é o compilador TurboC, que foi desenvolvido para o sistema operacional DOS mas ainda pode ser utilizado no Windows. No TurboC, o tipo `int` é mapeado para 2 bytes.

A linguagem oferece ainda dois tipos básicos para a representação de números reais (ponto flutuante): `float` e `double`. A tabela abaixo compara estes dois tipos.

Tipo	Tamanho	Representatividade
<code>float</code>	4 bytes	$\pm 10^{-38}$ a 10^{38}
<code>double</code>	8 bytes	$\pm 10^{-308}$ a 10^{308}

Declaração de variáveis

Para armazenarmos um dado (valor) na memória do computador, devemos reservar o espaço correspondente ao tipo do dado a ser armazenado. A declaração de uma variável reserva um espaço na memória para armazenar um dado do tipo da variável e associa o nome da variável a este espaço de memória.

```
int a;           /* declara uma variável do tipo int */  
int b;           /* declara outra variável do tipo int */  
float c;         /* declara uma variável do tipo float */  
  
a = 5;           /* armazena o valor 5 em a */  
b = 10;          /* armazena o valor 10 em b */  
c = 5.3;         /* armazena o valor 5.3 em c */
```

A linguagem permite que variáveis de mesmo tipo sejam declaradas juntas. Assim, as duas primeiras declarações acima poderiam ser substituídas por:

```
int a, b;        /* declara duas variáveis do tipo int */
```

Uma vez declarada a variável, podemos armazenar valores nos respectivos espaços de memória. Estes valores devem ter o mesmo tipo da variável, conforme ilustrado acima. Não é possível, por exemplo, armazenar um número real numa variável do tipo `int`. Se fizermos:

```
int a;  
a = 4.3;         /* a variável armazenará o valor 4 */
```

será armazenada em `a` apenas a parte inteira do número real, isto é, 4. Alguns compiladores exibem uma advertência quando encontram este tipo de atribuição.

Em C, as variáveis podem ser inicializadas na declaração. Podemos, por exemplo, escrever:

```
int a = 5, b = 10;    /* declara e inicializa as variáveis */  
float c = 5.3;
```

Valores constantes

Em nossos códigos, usamos também valores constantes. Quando escrevemos a atribuição:

```
a = b + 123;
```

sendo `a` e `b` variáveis supostamente já declaradas, reserva-se um espaço para armazenar a constante 123. No caso, a constante é do tipo inteiro, então um espaço de quatro bytes (em geral) é reservado e o valor 123 é armazenado nele. A diferença básica em relação às variáveis, como os nomes dizem (variáveis e constantes), é que o valor armazenado numa área de constante não pode ser alterado.

As constantes também podem ser do tipo real. Uma constante real deve ser escrita com um ponto decimal ou valor de expoente. Sem nenhum sufixo, uma constante real é do tipo `double`. Se quisermos uma constante real do tipo `float`, devemos, a rigor, acrescentar o sufixo `F` ou `f`. Alguns exemplos de constantes reais são:

```
12.45      constante real do tipo double
1245e-2    constante real do tipo double
12.45F     constante real do tipo float
```

Alguns compiladores exibem uma advertência quando encontram o código abaixo:

```
float x;
...
x = 12.45;
```

pois o código, a rigor, armazena um valor `double` (12.45) numa variável do tipo `float`. Desde que a constante seja representável dentro de um `float`, não precisamos nos preocupar com este tipo de advertência.

Variáveis com valores indefinidos

Um dos erros comuns em programas de computador é o uso de variáveis cujos valores ainda estão indefinidos. Por exemplo, o trecho de código abaixo está errado, pois o valor armazenado na variável `b` está indefinido e tentamos usá-lo na atribuição a `c`. É comum dizermos que `b` tem “lixo”.

```
int a, b, c;
a = 2;
c = a + b;           /* ERRO: b tem "lixo" */
```

Alguns destes erros são óbvios (como o ilustrado acima) e o compilador é capaz de nos reportar uma advertência; no entanto, muitas vezes o uso de uma variável não definida fica difícil de ser identificado no código. Repetimos que é um erro comum em programas e uma razão para alguns programas funcionarem na parte da manhã e não funcionarem na parte da tarde (ou funcionarem durante o desenvolvimento e não funcionarem quando entregamos para nosso cliente!). Todos os erros em computação têm lógica. A razão de o programa poder funcionar uma vez e não funcionar outra é que, apesar de indefinido, o valor da variável existe. No nosso caso acima, pode acontecer que o valor armazenado na memória ocupada por `b` seja 0, fazendo com que o programa funcione. Por outro lado, pode acontecer de o valor ser -293423 e o programa não funcionar.

2.2. Operadores

A linguagem C oferece uma gama variada de operadores, entre binários e unários. Os operadores básicos são apresentados a seguir.

Operadores Aritméticos

Os operadores aritméticos binários são: +, -, *, / e o operador módulo %. Há ainda o operador unário -. A operação é feita na precisão dos operandos. Assim, a expressão $5/2$ resulta no valor 2, pois a operação de divisão é feita em precisão inteira, já que os dois operandos (5 e 2) são constantes inteiras. A divisão de inteiros trunca a parte fracionária, pois o valor resultante é sempre do mesmo tipo da expressão. Conseqüentemente, a expressão $5.0/2.0$ resulta no valor real 2.5 pois a operação é feita na precisão real (double, no caso).

O operador módulo, %, não se aplica a valores reais, seus operandos devem ser do tipo inteiro. Este operador produz o resto da divisão do primeiro pelo segundo operando. Como exemplo de aplicação deste operador, podemos citar o caso em que desejamos saber se o valor armazenado numa determinada variável inteira x é par ou ímpar. Para tanto, basta analisar o resultado da aplicação do operador %, aplicado à variável e ao valor dois.

$x \% 2$	se resultado for zero	\Rightarrow número é par
$x \% 2$	se resultado for um	\Rightarrow número é ímpar

Os operadores *, / e % têm precedência maior que os operadores + e -. O operador - unário tem precedência maior que *, / e %. Operadores com mesma precedência são avaliados da esquerda para a direita. Assim, na expressão:

$a + b * c / d$

executa-se primeiro a multiplicação, seguida da divisão, seguida da soma. Podemos utilizar parênteses para alterar a ordem de avaliação de uma expressão. Assim, se quisermos avaliar a soma primeiro, podemos escrever:

$(a + b) * c / d$

Uma tabela de precedência dos operadores da linguagem C é apresentada no final desta seção.

Operadores de atribuição

Na linguagem C, uma atribuição é uma expressão cujo valor resultante corresponde ao valor atribuído. Assim, da mesma forma que a expressão:

$5 + 3$

resulta no valor 8, a atribuição:

$a = 5$

resulta no valor 5 (além, é claro, de armazenar o valor 5 na variável `a`). Este tratamento das atribuições nos permite escrever comandos do tipo:

```
y = x = 5;
```

Neste caso, a ordem de avaliação é da direita para a esquerda. Assim, o computador avalia `x = 5`, armazenando 5 em `x`, e em seguida armazena em `y` o valor produzido por `x = 5`, que é 5. Portanto, ambos, `x` e `y`, recebem o valor 5.

A linguagem também permite utilizar os chamados operadores de atribuição compostos. Comandos do tipo:

```
i = i + 2;
```

em que a variável à esquerda do sinal de atribuição também aparece à direita, podem ser escritas de forma mais compacta:

```
i += 2;
```

usando o operador de atribuição composto `+=`. Analogamente, existem, entre outros, os operadores de atribuição: `-=`, `*=`, `/=`, `%=`. De forma geral, comandos do tipo:

```
var op= expr;
```

são equivalentes a:

```
var = var op (expr);
```

Salientamos a presença dos parênteses em torno de `expr`. Assim:

```
x *= y + 1;
```

equivale a

```
x = x * (y + 1)
```

e não a

```
x = x * y + 1;
```

Operadores de incremento e decremento

A linguagem C apresenta ainda dois operadores não convencionais. São os operadores de incremento e decremento, que possuem precedência comparada ao - unário e servem para incrementar e decrementar uma unidade nos valores armazenados nas variáveis. Assim, se `n` é uma variável que armazena um valor, o comando:

```
n++;
```

incrementa de uma unidade o valor de n (análogo para o decremento em $n--$). O aspecto não usual é que $++$ e $--$ podem ser usados tanto como operadores pré-fixados (antes da variável, como em $++n$) ou pós-fixados (após a variável, como em $n++$). Em ambos os casos, a variável n é incrementada. Porém, a expressão $++n$ incrementa n *antes* de usar seu valor, enquanto $n++$ incrementa n *após* seu valor ser usado. Isto significa que, num contexto onde o valor de n é usado, $++n$ e $n++$ são diferentes. Se n armazena o valor 5, então:

```
x = n++;
```

atribui 5 a x , mas:

```
x = ++n;
```

atribuiria 6 a x . Em ambos os casos, n passa a valer 6, pois seu valor foi incrementado em uma unidade. Os operadores de incremento e decremento podem ser aplicados somente em variáveis; uma expressão do tipo $x = (i + 1)++$ é ilegal.

A linguagem C oferece diversas formas compactas para escrevermos um determinado comando. Neste curso, procuraremos evitar as formas compactas pois elas tendem a dificultar a compreensão do código. Mesmo para programadores experientes, o uso das formas compactas deve ser feito com critério. Por exemplo, os comandos:

```
a = a + 1;  
a += 1;  
a++;  
++a;
```

são todos equivalentes e o programador deve escolher o que achar mais adequado e simples. Em termos de desempenho, qualquer compilador razoável é capaz de otimizar todos estes comandos da mesma forma.

Operadores relacionais e lógicos

Os operadores relacionais em C são:

<	<i>menor que</i>
>	<i>maior que</i>
<=	<i>menor ou igual que</i>
>=	<i>maior ou igual que</i>
==	<i>igual a</i>
!=	<i>diferente de</i>

Estes operadores comparam dois valores. O resultado produzido por um operador relacional é zero ou um. Em C, não existe o tipo booleano (*true* ou *false*). O valor zero é interpretado como falso e qualquer valor diferente de zero é considerado verdadeiro. Assim, se o

resultado de uma comparação for falso, produz-se o valor 0, caso contrário, produz-se o valor 1.

Os operadores lógicos combinam expressões booleanas. A linguagem oferece os seguintes operadores lógicos:

&&	<i>operador binário E (AND)</i>
	<i>operador binário OU (OR)</i>
!	<i>operador unário de NEGAÇÃO (NOT)</i>

Expressões conectadas por && ou || são avaliadas da esquerda para a direita, e a avaliação pára assim que a veracidade ou falsidade do resultado for conhecida. Recomendamos o uso de parênteses em expressões que combinam operadores lógicos e relacionais.

Os operadores relacionais e lógicos são normalmente utilizados para tomada de decisões. No entanto, podemos utilizá-los para atribuir valores a variáveis. Por exemplo, o trecho de código abaixo é válido e armazena o valor 1 em a e 0 em b.

```
int a, b;
int c = 23;
int d = c + 4;

a = (c < 20) || (d > c);    /* verdadeiro */
b = (c < 20) && (d > c);   /* falso */
```

Devemos salientar que, na avaliação da expressão atribuída à variável b, a operação ($d > c$) não chega a ser avaliada, pois independente do seu resultado a expressão como um todo terá como resultado 0 (falso), uma vez que a operação ($c < 20$) tem valor falso.

Operador `sizeof`

Outro operador fornecido por C, `sizeof`, resulta no número de bytes de um determinado tipo. Por exemplo:

```
int a = sizeof(float);
```

armazena o valor 4 na variável a, pois um float ocupa 4 bytes de memória. Este operador pode também ser aplicado a uma variável, retornando o número de bytes do tipo associado à variável.

Conversão de tipo

Em C, como na maioria das linguagens, existem conversões automáticas de valores na avaliação de uma expressão. Assim, na expressão 3/1.5, o valor da constante 3 (tipo int) é promovido (convertido) para double antes de a expressão ser avaliada, pois o segundo operando é do tipo double (1.5) e a operação é feita na precisão do tipo mais representativo.

Quando, numa atribuição, o tipo do valor atribuído é diferente do tipo da variável, também há uma conversão automática de tipo. Por exemplo, se escrevermos:

```
int a = 3.5;
```

o valor 3.5 é convertido para inteiro (isto é, passa a valer 3) antes de a atribuição ser efetuada. Como resultado, como era de se esperar, o valor atribuído à variável é 3 (inteiro). Alguns compiladores exibem advertências quando a conversão de tipo pode significar uma perda de precisão (é o caso da conversão real para inteiro, por exemplo).

O programador pode explicitamente requisitar uma conversão de tipo através do uso do operador de molde de tipo (operador *cast*). Por exemplo, são válidos (e isentos de qualquer advertência por parte dos compiladores) os comandos abaixo.

```
int a, b;
a = (int) 3.5;
b = (int) 3.5 % 2;
```

Precedência e ordem de avaliação dos operadores

A tabela abaixo mostra a precedência, em ordem decrescente, dos principais operadores da linguagem C.

Operador	Associatividade
() [] -> .	esquerda para direita
! ~ ++ -- (tipo) * & sizeof(tipo)	direita para esquerda
* / %	esquerda para direita
+ -	esquerda para direita
<< >>	esquerda para direita
< <= > >=	esquerda para direita
== !=	esquerda para direita
&	esquerda para direita
^	esquerda para direita
	esquerda para direita
&&	esquerda para direita
	esquerda para direita
? :	direita para esquerda
= += -= etc.	direita para esquerda
,	esquerda para direita

2.3. Entrada e saída básicas

A linguagem C não possui comandos de entrada e saída do tipo READ e WRITE encontrados na linguagem FORTRAN. Tudo em C é feito através de funções, inclusive as operações de entrada e saída. Por isso, já existe em C uma biblioteca padrão que possui as funções básicas normalmente necessárias. Na biblioteca padrão de C, podemos, por exemplo, encontrar funções matemáticas do tipo raiz quadrada, seno, cosseno, etc., funções para a manipulação de cadeias de caracteres e funções de entrada e saída. Nesta seção, serão apresentadas as duas funções básicas de entrada e saída disponibilizadas pela biblioteca padrão. Para utilizá-las, é necessário incluir o *protótipo* destas funções no

código. Este assunto será tratado em detalhes na seção sobre funções. Por ora, basta saber que é preciso escrever:

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

no início do programa que utiliza as funções da biblioteca de entrada e saída.

Função printf

A função `printf` possibilita a saída de valores (sejam eles constantes, variáveis ou resultado de expressões) segundo um determinado formato. Informalmente, podemos dizer que a forma da função é:

```
printf (formato, lista de constantes/variáveis/expressões...) ;
```

O primeiro parâmetro é uma cadeia de caracteres, em geral delimitada com aspas, que especifica o formato de saída das constantes, variáveis e expressões listadas em seguida. Para cada valor que se deseja imprimir, deve existir um especificador de formato correspondente na cadeia de caracteres *formato*. Os especificadores de formato variam com o tipo do valor e a precisão em que queremos que eles sejam impressos. Estes especificadores são precedidos pelo caractere % e podem ser, entre outros:

%c	<i>especifica um char</i>
%d	<i>especifica um int</i>
%u	<i>especifica um unsigned int</i>
%f	<i>especifica um double (ou float)</i>
%e	<i>especifica um double (ou float) no formato científico</i>
%g	<i>especifica um double (ou float) no formato mais apropriado (%f ou %e)</i>
%s	<i>especifica uma cadeia de caracteres</i>

Alguns exemplos:

```
printf ("%d %g\n", 33, 5.3);
```

tem como resultado a impressão da linha:

```
33 5.3
```

Ou:

```
printf ("Inteiro = %d    Real = %g\n", 33, 5.3);
```

com saída:

```
Inteiro = 33    Real = 5.3
```

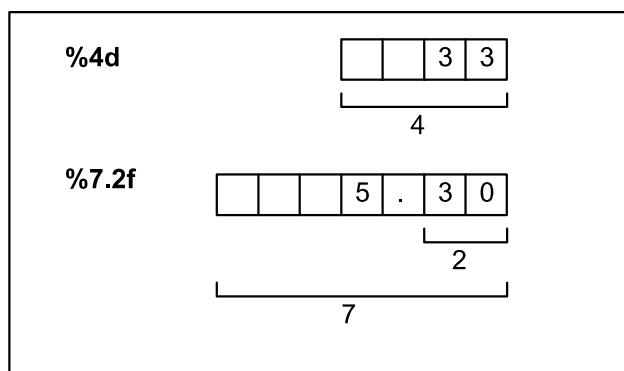
Isto é, além dos especificadores de formato, podemos incluir textos no formato, que são mapeados diretamente para a saída. Assim, a saída é formada pela cadeia de caracteres do formato onde os especificadores são substituídos pelos valores correspondentes.

Existem alguns caracteres de escape que são freqüentemente utilizados nos formatos de saída. São eles:

\n	<i>caractere de nova linha</i>
\t	<i>caractere de tabulação</i>
\r	<i>caractere de retrocesso</i>
\"	<i>o caractere "</i>
\\	<i>o caractere \</i>

Ainda, se desejarmos ter como saída um caractere %, devemos, dentro do formato, escrever %%.

É possível também especificarmos o tamanho dos campos:



A função `printf` retorna o número de campos impressos. Salientamos que para cada constante, variável ou expressão listada devemos ter um especificador de formato apropriado.

Função `scanf`

A função `scanf` permite capturarmos valores fornecidos via teclado pelo usuário do programa. Informalmente, podemos dizer que sua forma geral é:

```
scanf (formato, lista de endereços das variáveis...);
```

O formato deve possuir especificadores de tipos similares aos mostrados para a função `printf`. Para a função `scanf`, no entanto, existem especificadores diferentes para o tipo `float` e o tipo `double`:

<code>%c</code>	<i>especifica um char</i>
<code>%d</code>	<i>especifica um int</i>
<code>%u</code>	<i>especifica um unsigned int</i>
<code>%f, %e, %g</code>	<i>especificam um float</i>
<code>%lf, %le, %lg</code>	<i>especificam um double</i>
<code>%s</code>	<i>especifica uma cadeia de caracteres</i>

A principal diferença é que o formato deve ser seguido por uma lista de endereços de variáveis (na função `printf` passamos os valores de constantes, variáveis e expressões). Na seção sobre ponteiros, este assunto será tratado em detalhes. Por ora, basta saber que, para ler um valor e atribuí-lo a uma variável, devemos passar o endereço da variável para a função `scanf`. O operador `&` retorna o endereço de uma variável. Assim, para ler um inteiro, devemos ter:

```
int n;
scanf ("%d", &n);
```

Para a função `scanf`, os especificadores `%f`, `%e` e `%g` são equivalentes. Aqui, caracteres diferentes dos especificadores no formato servem para cercar a entrada. Por exemplo:

```
scanf ("%d:%d", &h, &m);
```

obriga que os valores (inteiros) fornecidos sejam separados pelo caractere dois pontos (:). Um espaço em branco dentro do formato faz com que sejam "pulados" eventuais brancos da entrada. Os especificadores `%d`, `%f`, `%e` e `%g` automaticamente pulam os brancos que precederem os valores numéricos a serem capturados. A função `scanf` retorna o número de campos lidos com sucesso.

3. Controle de fluxo

W. Celes e J. L. Rangel

A linguagem C provê as construções fundamentais de controle de fluxo necessárias para programas bem estruturados: agrupamentos de comandos, tomadas de decisão (`if-else`), laços com teste de encerramento no início (`while`, `for`) ou no fim (`do-while`), e seleção de um dentre um conjunto de possíveis casos (`switch`).

3.1. Decisões com `if`

`if` é o comando de decisão básico em C. Sua forma pode ser:

```
if (expr) {  
    bloco de comandos 1  
    ...  
}
```

ou

```
if ( expr ) {  
    bloco de comandos 1  
    ...  
}  
else {  
    bloco de comandos 2  
    ...  
}
```

Se `expr` produzir um valor diferente de 0 (verdadeiro), o *bloco de comandos 1* será executado. A inclusão do `else` requisita a execução do *bloco de comandos 2* se a expressão produzir o valor 0 (falso). Cada bloco de comandos deve ser delimitado por uma chave aberta e uma chave fechada. Se dentro de um bloco tivermos apenas um comando a ser executado, as chaves podem ser omitidas (na verdade, deixamos de ter um bloco):

```
if ( expr )  
    comando1;  
else  
    comando2;
```

A indentação (recurso de linha) dos comandos é fundamental para uma maior clareza do código. O estilo de indentação varia a gosto do programador. Além da forma ilustrada acima, outro estilo bastante utilizado por programadores C é:

```
if ( expr )  
{  
    bloco de comandos 1  
    ...  
}  
else  
{  
    bloco de comandos 2  
    ...  
}
```

Podemos aninhar comandos `if`. Um exemplo simples é ilustrado a seguir:

```
#include <stdio.h>

int main (void)
{
    int a, b;
    printf("Insira dois numeros inteiros:");
    scanf("%d%d", &a, &b);
    if (a%2 == 0)
        if (b%2 == 0)
            printf("Voce inseriu dois numeros pares!\n");
    return 0;
}
```

Primeiramente, notamos que não foi necessário criar blocos (`{...}`) porque a cada `if` está associado apenas um comando. Ao primeiro, associamos o segundo comando `if`, e ao segundo `if` associamos o comando que chama a função `printf`. Assim, o segundo `if` só será avaliado se o primeiro valor fornecido for par, e a mensagem só será impressa se o segundo valor fornecido também for par. Outra construção para este mesmo exemplo simples pode ser:

```
int main (void)
{
    int a, b;
    printf("Digite dois numeros inteiros:");
    scanf("%d%d", &a, &b);
    if ((a%2 == 0) && (b%2 == 0))
        printf ("Voce digitou dois numeros pares!\n");
    return 0;
}
```

produzindo resultados idênticos.

Devemos, todavia, ter cuidado com o aninhamento de comandos `if-else`. Para ilustrar, consideremos o exemplo abaixo.

```
/* temperatura (versao 1 - incorreta) */
#include <stdio.h>

int main (void)
{
    int temp;
    printf("Digite a temperatura: ");
    scanf("%d", &temp);
    if (temp < 30)
        if (temp > 20)
            printf(" Temperatura agradavel \n");
        else
            printf(" Temperatura muito quente \n");
    return 0;
}
```

A idéia deste programa era imprimir a mensagem Temperatura agradável se fosse fornecido um valor entre 20 e 30, e imprimir a mensagem Temperatura muito quente se fosse fornecido um valor maior que 30. No entanto, vamos analisar o caso de

ser fornecido o valor 5 para `temp`. Observando o código do programa, podemos pensar que nenhuma mensagem seria fornecida, pois o primeiro `if` daria resultado verdadeiro e então seria avaliado o segundo `if`. Neste caso, teríamos um resultado falso e como, *aparentemente*, não há um comando `else` associado, nada seria impresso. Puro engano. A indentação utilizada pode nos levar a erros de interpretação. O resultado para o valor 5 seria a mensagem `Temperatura muito quente`. Isto é, o programa está INCORRETO.

Em C, um `else` está associado ao último `if` que não tiver seu próprio `else`. Para os casos em que a associação entre `if` e `else` não está clara, recomendamos a criação explícita de blocos, mesmo contendo um único comando. Reescrevendo o programa, podemos obter o efeito desejado.

```
/* temperatura (versao 2) */
#include <stdio.h>

int main (void)
{
    int temp;
    printf ( "Digite a temperatura: " );
    scanf ( "%d", &temp );
    if ( temp < 30 )
    {
        if ( temp > 20 )
            printf ( " Temperatura agradavel \n" );
    }
    else
        printf ( " Temperatura muito quente \n" );
    return 0;
}
```

Esta regra de associação do `else` propicia a construção do tipo `else-if`, sem que se tenha o comando `elseif` explicitamente na gramática da linguagem. Na verdade, em C, construímos estruturas `else-if` com `if`'s aninhados. O exemplo abaixo é válido e funciona como esperado.

```
/* temperatura (versao 3) */
#include <stdio.h>

int main (void)
{
    int temp;
    printf("Digite a temperatura: ");
    scanf("%d", &temp);

    if (temp < 10)
        printf("Temperatura muito fria \n");
    else if (temp < 20)
        printf(" Temperatura fria \n");
    else if (temp < 30)
        printf("Temperatura agradavel \n");
    else
        printf("Temperatura muito quente \n");
    return 0;
}
```

Estruturas de bloco

Observamos que uma função C é composta por estruturas de blocos. Cada chave aberta e fechada em C representa um bloco. As declarações de variáveis só podem ocorrer no início do corpo da função ou no início de um bloco, isto é, devem seguir uma chave aberta. Uma variável declarada dentro de um bloco é válida apenas dentro do bloco. Após o término do bloco, a variável deixa de existir. Por exemplo:

```
...
if ( n > 0 )
{
    int i;
    ...
}
...           /* a variável i não existe neste ponto do programa */
```

A variável `i`, definida dentro do bloco do `if`, só existe dentro deste bloco. É uma boa prática de programação declarar as varáveis o mais próximo possível dos seus usos.

Operador condicional

C possui também um chamado *operador condicional*. Trata-se de um operador que substitui construções do tipo:

```
...
if ( a > b )
    maximo = a;
else
    maximo = b;
...
```

Sua forma geral é:

condição ? *expressão1* : *expressão2*;

se a *condição* for verdadeira, a *expressão1* é avaliada; caso contrário, avalia-se a *expressão2*.

O comando:

```
maximo = a > b ? a : b ;
```

substitui a construção com `if-else` mostrada acima.

3.2. Construções com laços

É muito comum, em programas computacionais, termos procedimentos iterativos, isto é, procedimentos que devem ser executados em vários passos. Como exemplo, vamos considerar o cálculo do valor do fatorial de um número inteiro não negativo. Por definição:

$$n! = n \times (n-1) \times (n-2) \dots 3 \times 2 \times 1, \text{ onde } 0! = 1$$

Para calcular o fatorial de um número através de um programa de computador, utilizamos tipicamente um processo iterativo, em que o valor da variável varia de 1 a n.

A linguagem C oferece diversas construções possíveis para a realização de laços iterativos. O primeiro a ser apresentado é o comando `while`. Sua forma geral é:

```
while (expr)
{
    bloco de comandos
    ...
}
```

Enquanto *expr* for avaliada em verdadeiro, o bloco de comandos é executado repetidamente. Se *expr* for avaliada em falso, o bloco de comando não é executado e a execução do programa prossegue. Uma possível implementação do cálculo do fatorial usando `while` é mostrada a seguir.

```
/* Fatorial */
#include <stdio.h>

int main (void)
{
    int i;
    int n;
    int f = 1;

    printf("Digite um número inteiro não negativo:");
    scanf("%d", &n);

    /* calcula fatorial */
    i = 1;
    while (i <= n)
    {
        f *= i;
        i++;
    }

    printf(" Fatorial = %d \n", f);
    return 0;
}
```

Uma segunda forma de construção de laços em C, mais compacta e amplamente utilizada, é através de laços `for`. A forma geral do `for` é:

```
for (expr_inicial; expr_booleana; expr_de_incremento)
{
    bloco de comandos
    ...
}
```

A ordem de avaliação desta construção é ilustrada a seguir:

```
expr_inicial;
while (expr_booleana)
{
    bloco de comandos
    ...
    expr_de_incremento
}
```

A seguir, ilustramos a utilização do comando `for` no programa para cálculo do fatorial.

```
/* Fatorial (versao 2) */
#include <stdio.h>

int main (void)
{
    int i;
    int n;
    int f = 1;

    printf("Digite um número inteiro não negativo:");
    scanf("%d", &n);

    /* calcula fatorial */
    for (i = 1; i <= n; i++)
    {
        f *= i;
    }
    printf(" Fatorial = %d \n", f);
    return 0;
}
```

Observamos que as chaves que seguem o comando `for`, neste caso, são desnecessárias, já que o corpo do bloco é composto por um único comando.

Tanto a construção com `while` como a construção com `for` avaliam a expressão booleana que caracteriza o teste de encerramento no início do laço. Assim, se esta expressão tiver valor igual a zero (falso), quando for avaliada pela primeira vez, os comandos do corpo do bloco não serão executados nem uma vez.

C provê outro comando para construção de laços cujo teste de encerramento é avaliado no final. Esta construção é o `do-while`, cuja forma geral é:

```
do
{
    bloco de comandos
} while (expr_booleana);
```

Um exemplo do uso desta construção é mostrado abaixo, onde validamos a inserção do usuário, isto é, o programa repetidamente requisita a inserção de um número enquanto o usuário inserir um inteiro negativo (cujo fatorial não está definido).

```

/* Fatorial (versao 3) */

#include <stdio.h>

int main (void)
{
    int i;
    int n;
    int f = 1;

    /* requisita valor do usuário */
    do
    {
        printf("Digite um valor inteiro não negativo:");
        scanf ("%d", &n);
    } while (n<0);

    /* calcula fatorial */
    for (i = 1; i <= n; i++)
        f *= i;

    printf(" Fatorial = %d\n", f);
    return 0;
}

```

Interrupções com break e continue

A linguagem C oferece ainda duas formas para a interrupção antecipada de um determinado laço. O comando `break`, quando utilizado dentro de um laço, interrompe e termina a execução do mesmo. A execução prossegue com os comandos subsequentes ao bloco. O código abaixo ilustra o efeito de sua utilização.

```

#include <stdio.h>

int main (void)
{
    int i;
    for (i = 0; i < 10; i++)
    {
        if (i == 5)
            break;
        printf("%d ", i);
    }
    printf("fim\n");
    return 0;
}

```

A saída deste programa, se executado, será:

```
0 1 2 3 4  fim
```

pois, quando `i` tiver o valor 5, o laço será interrompido e finalizado pelo comando `break`, passando o controle para o próximo comando após o laço, no caso uma chamada final de `printf`.

O comando `continue` também interrompe a execução dos comandos de um laço. A diferença básica em relação ao comando `break` é que o laço não é automaticamente finalizado. O comando `continue` interrompe a execução de um laço passando para a próxima iteração. Assim, o código:

```
#include <stdio.h>

int main (void)
{
    int i;
    for (i = 0; i < 10; i++)
    {
        if (i == 5) continue;
        printf("%d ", i);
    }
    printf("fim\n");
    return 0;
}
```

gera a saída:

```
0 1 2 3 4 6 7 8 9  fim
```

Devemos ter cuidado com a utilização do comando `continue` nos laços `while`. O programa:

```
/* INCORRETO */

#include <stdio.h>

int main (void)
{
    int i = 0;
    while (i < 10)
    {
        if (i == 5) continue;
        printf("%d ", i);
        i++;
    }
    printf("fim\n");
    return 0;
}
```

é um programa INCORRETO, pois o laço criado não tem fim – a execução do programa não termina. Isto porque a variável `i` nunca terá valor superior a 5, e o teste será sempre verdadeiro. O que ocorre é que o comando `continue` "pula" os demais comandos do laço quando `i` vale 5, inclusive o comando que incrementa a variável `i`.

3.3. Seleção

Além da construção `else-if`, C provê um comando (`switch`) para selecionar um dentre um conjunto de possíveis casos. Sua forma geral é:

```

switch ( expr )
{
    case op1:
        ...      /* comandos executados se expr == op1 */
        break;
    case op2:
        ...      /* comandos executados se expr == op2 */
        break;
    case op3:
        ...      /* comandos executados se expr == op3 */
        break;
    default:
        ...      /* executados se expr for diferente de todos */
        break;
}

```

op_i deve ser um número inteiro ou uma constante caractere. Se $expr$ resultar no valor op_i , os comandos que se seguem ao caso op_i são executados, até que se encontre um `break`. Se o comando `break` for omitido, a execução do caso continua com os comandos do caso seguinte. O caso `default` (nenhum dos outros) pode aparecer em qualquer posição, mas normalmente é colocado por último. Para exemplificar, mostramos a seguir um programa que implementa uma calculadora convencional que efetua as quatro operações básicas. Este programa usa constantes caracteres, que serão discutidas em detalhe quando apresentarmos cadeias de caracteres em C. O importante aqui é entender conceitualmente a construção `switch`.

```

/* calculadora de quatro operações */

#include <stdio.h>

int main (void)
{
    float num1, num2;
    char op;

    printf("Digite: numero op numero\n");
    scanf ("%f %c %f", &num1, &op, &num2);
    switch (op)
    {
        case '+':
            printf(" = %f\n", num1+num2);
            break;
        case '-':
            printf(" = %f\n", num1-num2);
            break;
        case '*':
            printf(" = %f\n", num1*num2);
            break;
        case '/':
            printf(" = %f\n", num1/num2);
            break;
        default:
            printf("Operador invalido!\n");
            break;
    }
    return 0;
}

```

4. Funções

W. Celes e J. L. Rangel

4.1. Definição de funções

As funções dividem grandes tarefas de computação em tarefas menores. Os programas em C geralmente consistem de várias pequenas funções em vez de poucas de maior tamanho. A criação de funções evita a repetição de código, de modo que um procedimento que é repetido deve ser transformado numa função que, então, será chamada diversas vezes. Um programa bem estruturado deve ser pensado em termos de funções, e estas, por sua vez, podem (e devem, se possível) esconder do corpo principal do programa detalhes ou particularidades de implementação. Em C, tudo é feito através de funções. Os exemplos anteriores utilizam as funções da biblioteca padrão para realizar entrada e saída. Neste capítulo, discutiremos a codificação de nossas próprias funções.

A forma geral para definir uma função é:

```
tipo_retornado nome_da_função (lista de parâmetros...)
{
    corpo da função
}
```

Para ilustrar a criação de funções, consideraremos o cálculo do fatorial de um número. Podemos escrever uma função que, dado um determinado número inteiro não negativo n , imprime o valor de seu fatorial. Um programa que utiliza esta função seria:

```
/* programa que le um numero e imprime seu fatorial */
#include <stdio.h>

void fat (int n);

/* Função principal */
int main (void)
{
    int n;
    scanf("%d", &n);
    fat(n);
    return 0;
}

/* Função para imprimir o valor do fatorial */
void fat ( int n )
{
    int i;
    int f = 1;
    for (i = 1; i <= n; i++)
        f *= i;
    printf("Fatorial = %d\n", f);
}
```

Notamos, neste exemplo, que a função `fat` recebe como parâmetro o número cujo fatorial deve ser impresso. Os parâmetros de uma função devem ser listados, com seus respectivos tipos, entre os parênteses que seguem o nome da função. Quando uma função não tem parâmetros, colocamos a palavra reservada `void` entre os parênteses. Devemos notar que `main` também é uma função; sua única particularidade consiste em ser a função automaticamente executada após o programa ser carregado. Como as funções `main` que temos apresentado não recebem parâmetros, temos usado a palavra `void` na lista de parâmetros.

Além de receber parâmetros, uma função pode ter um valor de retorno associado. No exemplo do cálculo do fatorial, a função `fat` não tem nenhum valor de retorno, portanto colocamos a palavra `void` antes do nome da função, indicando a ausência de um valor de retorno.

```
void fat (int n)
{
    . .
}
```

A função `main` obrigatoriamente deve ter um valor inteiro como retorno. Esse valor pode ser usado pelo sistema operacional para testar a execução do programa. A convenção geralmente utilizada faz com que a função `main` retorne zero no caso da execução ser bem sucedida ou diferente de zero no caso de problemas durante a execução.

Por fim, salientamos que C exige que se coloque o *protótipo* da função antes desta ser chamada. O *protótipo* de uma função consiste na repetição da linha de sua definição seguida do caractere `(;)`. Temos então:

```
void fat (int n);      /* obs: existe ; no protótipo */
int main (void)
{
    . .
}
void fat (int n)      /* obs: não existe ; na definição */
{
    . .
}
```

A rigor, no protótipo não há necessidade de indicarmos os nomes dos parâmetros, apenas os seus tipos, portanto seria válido escrever: `void fat (int);`. Porém, geralmente mantemos os nomes dos parâmetros, pois servem como documentação do significado de cada parâmetro, desde que utilizemos nomes coerentes. O protótipo da função é necessário para que o compilador verifique os tipos dos parâmetros na chamada da função. Por exemplo, se tentássemos chamar a função com `fat(4.5)`; o compilador provavelmente indicaria o erro, pois estariam passando um valor real enquanto a função espera um valor inteiro. É devido a esta necessidade que se exige a inclusão do arquivo `stdio.h` para a utilização das funções de entrada e saída da biblioteca padrão. Neste arquivo, encontram-se, entre outras coisas, os protótipos das funções `printf` e `scanf`.

Uma função pode ter um valor de retorno associado. Para ilustrar a discussão, vamos reescrever o código acima, fazendo com que a função `fat` retorne o valor do fatorial. A função `main` fica então responsável pela impressão do valor.

```
/* programa que le um numero e imprime seu fatorial (versão 2) */
#include <stdio.h>

int fat (int n);

int main (void)
{
    int n, r;
    scanf("%d", &n);
    r = fat(n);
    printf("Fatorial = %d\n", r);
    return 0;
}

/* função para calcular o valor do fatorial */
int fat (int n)
{
    int i;
    int f = 1;
    for (i = 1; i <= n; i++)
        f *= i;
    return f;
}
```

4.2. Pilha de execução

Apresentada a forma básica para a definição de funções, discutiremos agora, em detalhe, como funciona a comunicação entre a função que chama e a função que é chamada. Já mencionamos na introdução deste curso que as funções são independentes entre si. As variáveis locais definidas dentro do corpo de uma função (e isto inclui os parâmetros das funções) não existem fora da função. Cada vez que a função é executada, as variáveis locais são criadas, e, quando a execução da função termina, estas variáveis deixam de existir.

A transferência de dados entre funções é feita através dos parâmetros e do valor de retorno da função chamada. Conforme mencionado, uma função pode retornar um valor para a função que a chamou e isto é feito através do comando `return`. Quando uma função tem um valor de retorno, a chamada da função é uma expressão cujo valor resultante é o valor retornado pela função. Por isso, foi válido escrevermos na função `main` acima a expressão `r = fat(n);` que chama a função `fat` armazenando seu valor de retorno na variável `r`.

A comunicação através dos parâmetros requer uma análise mais detalhada. Para ilustrar a discussão, vamos considerar o exemplo abaixo, no qual a implementação da função `fat` foi ligeiramente alterada:

```

/* programa que le um numero e imprime seu fatorial (versão 3) */

#include <stdio.h>

int fat (int n);

int main (void)
{
    int n = 5;
    int r;
    r = fat (n);
    printf("Fatorial de %d = %d \n", n, r);
    return 0;
}

int fat (int n)
{
    int f = 1.0;
    while (n != 0)
    {
        f *= n;
        n--;
    }
    return f;
}

```

Neste exemplo, podemos verificar que, no final da função `fat`, o parâmetro `n` tem valor igual a zero (esta é a condição de encerramento do laço `while`). No entanto, a saída do programa será:

```
Fatorial de 5 = 120
```

pois o valor da variável `n` não mudou no programa principal. Isto porque a linguagem C trabalha com o conceito de **passagem por valor**. Na chamada de uma função, o valor passado é atribuído ao parâmetro da função chamada. Cada parâmetro funciona como uma variável local inicializada com o valor passado na chamada. Assim, a variável `n` (parâmetro da função `fat`) é local e não representa a variável `n` da função `main` (o fato de as duas variáveis terem o mesmo nome é indiferente; poderíamos chamar o parâmetro de `v`, por exemplo). Alterar o valor de `n` dentro de `fat` não afeta o valor da variável `n` de `main`.

A execução do programa funciona com o **modelo de pilha**. De forma simplificada, o modelo de pilha funciona da seguinte maneira: cada variável local de uma função é colocada na pilha de execução. Quando se faz uma chamada a uma função, os parâmetros são copiados para a pilha e são tratados como se fossem variáveis locais da função chamada. Quando a função termina, a parte da pilha correspondente àquela função é liberada, e por isso não podemos acessar as variáveis locais de fora da função em que elas foram definidas.

Para exemplificar, vamos considerar um esquema representativo da memória do computador. Salientamos que este esquema é apenas uma maneira didática de explicar o que ocorre na memória do computador. Suponhamos que as variáveis são armazenadas na memória como ilustrado abaixo. Os números à direita representam endereços (posições)

fictícios de memória e os nomes à esquerda indicam os nomes das variáveis. A figura abaixo ilustra este esquema representativo da memória que adotaremos.

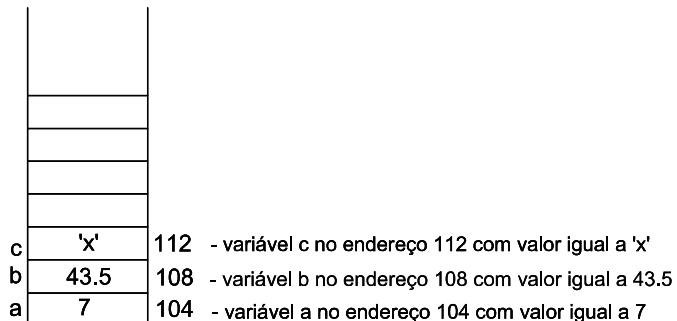


Figura 4.1: Esquema representativo da memória.

Podemos, então, analisar passo a passo a evolução do programa mostrado acima, ilustrando o funcionamento da pilha de execução.

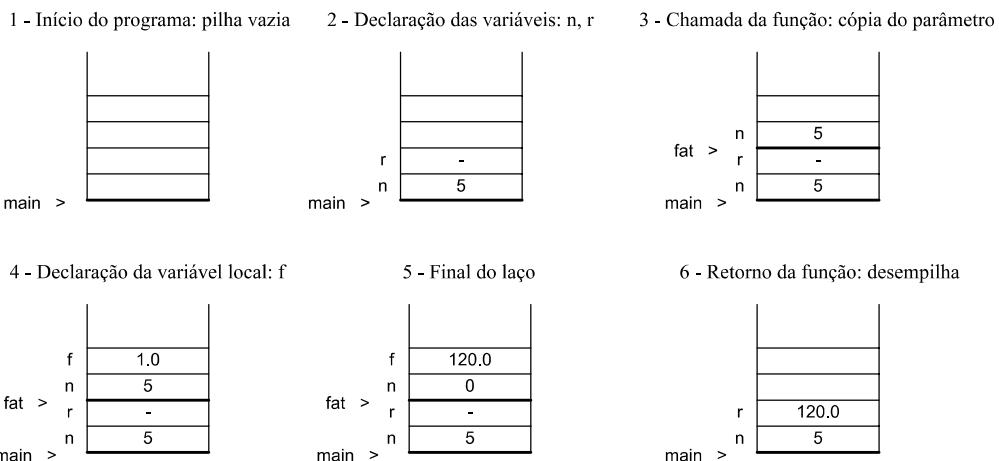


Figura 4.2: Execução do programa passo a passo.

Isto ilustra por que o valor da variável passada nunca será alterado dentro da função. A seguir, discutiremos uma forma para podermos alterar valores por passagem de parâmetros, o que será realizado passando o endereço de memória onde a variável está armazenada.

Vale salientar que existe outra forma de fazermos comunicação entre funções, que consiste no uso de variáveis globais. Se uma determinada variável global é visível em duas funções, ambas as funções podem acessar e/ou alterar o valor desta variável diretamente. No

entanto, conforme já mencionamos, o uso de variáveis globais em um programa deve ser feito com critério, pois podemos criar códigos com uma alto grau de interdependência entre as funções, o que dificulta a manutenção e o reuso do código.

4.3. Ponteiro de variáveis

A linguagem C permite o armazenamento e a manipulação de valores de endereços de memória. Para cada tipo existente, há um tipo ponteiro que pode armazenar endereços de memória onde existem valores do tipo correspondente armazenados. Por exemplo, quando escrevemos:

```
int a;
```

declaramos uma variável com nome `a` que pode armazenar valores inteiros. Automaticamente, reserva-se um espaço na memória suficiente para armazenar valores inteiros (geralmente 4 bytes).

Da mesma forma que declaramos variáveis para armazenar inteiros, podemos declarar variáveis que, em vez de servirem para armazenar valores inteiros, servem para armazenar valores de endereços de memória onde há variáveis inteiras armazenadas. C não reserva uma palavra especial para a declaração de ponteiros; usamos a mesma palavra do tipo com os nomes das variáveis precedidas pelo caractere `*`. Assim, podemos escrever:

```
int *p;
```

Neste caso, declaramos uma variável com nome `p` que pode armazenar endereços de memória onde existe um inteiro armazenado. Para atribuir e acessar endereços de memória, a linguagem oferece dois operadores unários ainda não discutidos. O operador unário `&` (“endereço de”), aplicado a variáveis, resulta no endereço da posição da memória reservada para a variável. O operador unário `*` (“conteúdo de”), aplicado a variáveis do tipo ponteiro, acessa o conteúdo do endereço de memória armazenado pela variável ponteiro. Para exemplificar, vamos ilustrar esquematicamente, através de um exemplo simples, o que ocorre na pilha de execução. Consideremos o trecho de código mostrado na figura abaixo.

```
/*variável inteiro */
int a;

/*variável ponteiro p/ inteiro */
int *p;
```

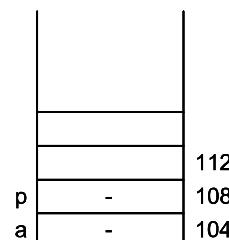
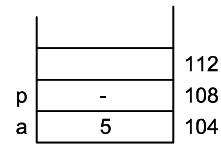


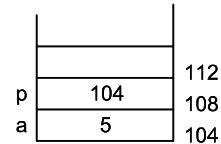
Figura 4.3: Efeito de declarações de variáveis na pilha de execução.

Após as declarações, ambas as variáveis, `a` e `p`, armazenam valores "lixo", pois não foram inicializadas. Podemos fazer atribuições como exemplificado nos fragmentos de código da figura a seguir:

```
/* a recebe o valor 5 */
a = 5;
```



```
/* p recebe o endereço de a
(diz-se p aponta para a) */
p = &a;
```



```
/* conteúdo de p recebe o valor 6 */
*p = 6;
```

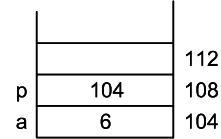


Figura 4.4: Efeito de atribuição de variáveis na pilha de execução.

Com as atribuições ilustradas na figura, a variável `a` recebe, indiretamente, o valor 6. Acessar `a` é equivalente a acessar `*p`, pois `p` armazena o endereço de `a`. Dizemos que `p` *aponta para a*, daí o nome *ponteiro*. Em vez de criarmos valores fictícios para os endereços de memória no nosso esquema ilustrativo da memória, podemos desenhar setas graficamente, sinalizando que um ponteiro aponta para uma determinada variável.

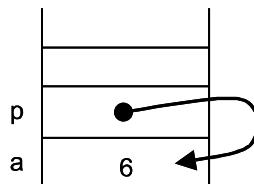


Figura 4.5: Representação gráfica do valor de um ponteiro.

A possibilidade de manipular ponteiros de variáveis é uma das maiores potencialidades de C. Por outro lado, o uso indevido desta manipulação é o maior causador de programas que "voam", isto é, não só não funcionam como, pior ainda, podem gerar efeitos colaterais não previstos.

A seguir, apresentamos outros exemplos de uso de ponteiros. O código abaixo:

```
int main ( void )
{
    int a;
    int *p;
    p = &a;
    *p = 2;
    printf(" %d ", a);
    return;
}
```

imprime o valor 2.

Agora, no exemplo abaixo:

```
int main ( void )
{
    int a, b, *p;
    a = 2;
    *p = 3;
    b = a + (*p);
    printf(" %d ", b);
    return 0;
}
```

cometemos um ERRO típico de manipulação de ponteiros. O pior é que esse programa, embora incorreto, às vezes pode funcionar. O erro está em usar a memória apontada por *p* para armazenar o valor 3. Ora, a variável *p* não tinha sido inicializada e, portanto, tinha armazenado um valor (no caso, endereço) "lixo". Assim, a atribuição **p = 3*; armazena 3 num espaço de memória desconhecido, que tanto pode ser um espaço de memória não utilizado, e aí o programa aparentemente funciona bem, quanto um espaço que armazena outras informações fundamentais – por exemplo, o espaço de memória utilizado por outras variáveis ou outros aplicativos. Neste caso, o erro pode ter efeitos colaterais indesejados.

Portanto, só podemos preencher o conteúdo de um ponteiro se este tiver sido devidamente inicializado, isto é, ele deve apontar para um espaço de memória onde já se prevê o armazenamento de valores do tipo em questão.

De maneira análoga, podemos declarar ponteiros de outros tipos:

```
float *m;
char *s;
```

Passando ponteiros para funções

Os ponteiros oferecem meios de alterarmos valores de variáveis acessando-as indiretamente. Já discutimos que as funções não podem alterar diretamente valores de variáveis da função que fez a chamada. No entanto, se passarmos para uma função os valores dos endereços de memória onde suas variáveis estão armazenadas, a função pode alterar, indiretamente, os valores das variáveis da função que a chamou.

Vamos analisar o uso desta estratégia através de um exemplo. Consideremos uma função projetada para trocar os valores entre duas variáveis. O código abaixo:

```
/* funcao troca (versao ERRADA) */
#include <stdio.h>

void troca (int x, int y )
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main ( void )
{
    int a = 5, b = 7;
    troca(a, b);
    printf("%d %d \n", a, b);
    return 0;
}
```

não funciona como esperado (serão impressos 5 e 7), pois os valores de `a` e `b` da função `main` não são alterados. Alterados são os valores de `x` e `y` dentro da função `troca`, mas eles não representam as variáveis da função `main`, apenas são inicializados com os valores de `a` e `b`. A alternativa é fazer com que a função receba os endereços das variáveis e, assim, alterar seus valores indiretamente. Reescrevendo:

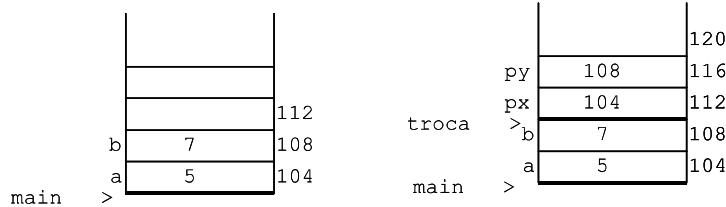
```
/* funcao troca (versao CORRETA) */
#include <stdio.h>

void troca (int *px, int *py )
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

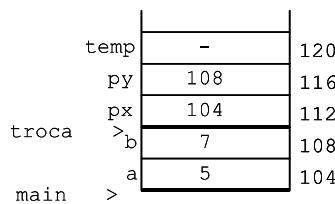
int main ( void )
{
    int a = 5, b = 7;
    troca(&a, &b);      /* passamos os endereços das variáveis */
    printf("%d %d \n", a, b);
    return 0;
}
```

A Figura 4.6 ilustra a execução deste programa mostrando o uso da memória. Assim, conseguimos o efeito desejado. Agora fica explicado por que passamos o endereço das variáveis para a função `scanf`, pois, caso contrário, a função não conseguiria devolver os valores lidos.

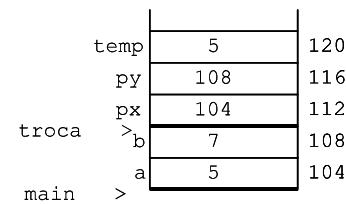
1 -Declaração das variáveis: a, b 2 - Chamada da função: passa endereços



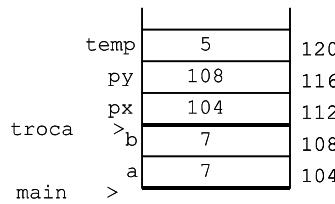
3 - Declaração da variável local: temp



4 - temp recebe conteúdo de px



5 -Conteúdo de px recebe conteúdo de py



6 -Conteúdo de py recebe temp

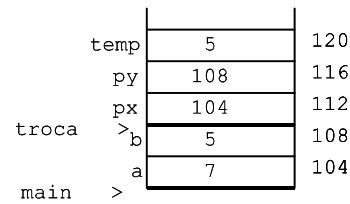


Figura 4.6: Passo a passo da função que troca dois valores.

4.4. Recursividade

As funções podem ser chamadas recursivamente, isto é, dentro do corpo de uma função podemos chamar novamente a própria função. Se uma função A chama a própria função A, dizemos que ocorre uma recursão direta. Se uma função A chama uma função B que, por sua vez, chama A, temos uma recursão indireta. Diversas implementações ficam muito mais fáceis usando recursividade. Por outro lado, implementações não recursivas tendem a ser mais eficientes.

Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução. Assim, mesmo quando uma função é chamada recursivamente, cria-se um ambiente local para cada chamada. As variáveis locais de chamadas recursivas são independentes entre si, como se estivéssemos chamando funções diferentes.

As implementações recursivas devem ser pensadas considerando-se a definição recursiva do problema que desejamos resolver. Por exemplo, o valor do fatorial de um número pode ser definido de forma recursiva:

$$n! = \begin{cases} 1, & \text{se } n = 0 \\ n \times (n-1)!, & \text{se } n > 0 \end{cases}$$

Considerando a definição acima, fica muito simples pensar na implementação recursiva de uma função que calcula e retorna o fatorial de um número.

```
/* Função recursiva para cálculo do fatorial */

int fat (int n)
{
    if (n==0)
        return 1;
    else
        return n*fat(n-1);
}
```

4.5. Variáveis estáticas dentro de funções**

Podemos declarar variáveis estáticas dentro de funções. Neste caso, as variáveis não são armazenadas na pilha, mas sim numa área de memória estática que existe enquanto o programa está sendo executado. Ao contrário das variáveis locais (ou automáticas), que existem apenas enquanto a função à qual elas pertencem estiver sendo executada, as estáticas, assim como as globais, continuam existindo mesmo antes ou depois de a função ser executada. No entanto, uma variável estática declarada dentro de uma função só é visível dentro dessa função. Uma utilização importante de variáveis estáticas dentro de funções é quando se necessita recuperar o valor de uma variável atribuída na última vez que a função foi executada.

Para exemplificar a utilização de variáveis estáticas declaradas dentro de funções, consideremos uma função que serve para imprimir números reais. A característica desta função é que ela imprime um número por vez, separando-os por espaços em branco e colocando, no máximo, cinco números por linha. Com isto, do primeiro ao quinto número são impressos na primeira linha, do sexto ao décimo na segunda, e assim por diante.

```
void imprime ( float a )
{
    static int n = 1;

    printf(" %f ", a);
    if ((n % 5) == 0) printf(" \n ");
    n++;
}
```

Se uma variável estática não for explicitamente inicializada na declaração, ela é automaticamente inicializada com zero. (As variáveis globais também são, por *default*, inicializadas com zero.)

4.6. Pré-processador e macros**

Um código C, **antes** de ser compilado, passa por um pré-processador. O pré-processador de C reconhece determinadas diretivas e altera o código para, então, enviá-lo ao compilador.

Uma das diretivas reconhecidas pelo pré-processador, e já utilizada nos nossos exemplos, é `#include`. Ela é seguida por um nome de arquivo e o pré-processador a substitui pelo corpo do arquivo especificado. É como se o texto do arquivo incluído fizesse parte do código fonte.

Uma observação: quando o nome do arquivo a ser incluído é envolto por aspas ("arquivo"), o pré-processador procura primeiro o arquivo no diretório atual e, caso não o encontre, o procura nos diretórios de *include* especificados para compilação. Se o arquivo é colocado entre os sinais de menor e maior (<arquivo>), o pré-processador não procura o arquivo no diretório atual.

Outra diretiva de pré-processamento que é muito utilizada e que será agora discutida é a diretiva de definição. Por exemplo, uma função para calcular a área de um círculo pode ser escrita da seguinte forma:

```
#define PI 3.14159

float area (float r)
{
    float a = PI * r * r;
    return a;
}
```

Neste caso, antes da compilação, toda ocorrência da palavra `PI` (desde que não envolvida por aspas) será trocada pelo número `3.14159`. O uso de diretivas de definição para representarmos constantes simbólicas é fortemente recomendável, pois facilita a manutenção e acrescenta clareza ao código. C permite ainda a utilização da diretiva de definição com parâmetros. É válido escrever, por exemplo:

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

assim, se após esta definição existir uma linha de código com o trecho:

```
v = 4.5;
c = MAX ( v, 3.0 );
```

o compilador verá:

```
v = 4.5;
c = ((v) > (4.5) ? (v) : (4.5));
```

Estas definições com parâmetros recebem o nome de **macros**. Devemos ter muito cuidado na definição de macros. Mesmo um erro de sintaxe pode ser difícil de ser detectado, pois o

compilador indicará um erro na linha em que se utiliza a macro e não na linha de definição da macro (onde efetivamente encontra-se o erro). Outros efeitos colaterais de macros mal definidas podem ser ainda piores. Por exemplo, no código abaixo:

```
#include <stdio.h>

#define DIF(a,b)    a - b

int main (void)
{
    printf(" %d ", 4 * DIF(5,3));
    return 0;
}
```

o resultado impresso é 17 e não 8, como poderia ser esperado. A razão é simples, pois para o compilador (fazendo a substituição da macro) está escrito:

```
printf(" %d ", 4 * 5 - 3);
```

e a multiplicação tem precedência sobre a subtração. Neste caso, parênteses envolvendo a macro resolveriam o problema. Porém, neste outro exemplo que envolve a macro com parênteses:

```
#include <stdio.h>

#define PROD(a,b)    (a * b)

int main (void)
{
    printf(" %d ", PROD(3+4, 2));
    return 0;
}
```

o resultado é 11 e não 14. A macro corretamente definida seria:

```
#define PROD(a,b)    ((a) * (b))
```

Concluímos, portanto, que, como regra básica para a definição de macros, devemos envolver cada parâmetro, e a macro como um todo, com parênteses.

5. Vetores e alocação dinâmica

W. Celes e J. L. Rangel

5.1. Vetores

A forma mais simples de estruturarmos um conjunto de dados é por meio de vetores. Como a maioria das linguagens de programação, C permite a definição de vetores. Definimos um vetor em C da seguinte forma:

```
int v[10];
```

A declaração acima diz que `v` é um vetor de inteiros dimensionado com 10 elementos, isto é, reservamos um espaço de memória **contínuo** para armazenar 10 valores inteiros. Assim, se cada `int` ocupa 4 bytes, a declaração acima reserva um espaço de memória de 40 bytes, como ilustra a figura abaixo.

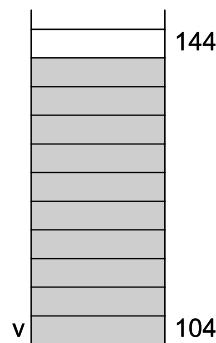


Figura 5.1: Espaço de memória de um vetor de 10 elementos inteiros.

O acesso a cada elemento do vetor é feito através de uma indexação da variável `v`. Observamos que, em C, a indexação de um vetor varia de zero a $n-1$, onde n representa a dimensão do vetor. Assim:

<code>v[0]</code>	→	acessa o primeiro elemento de <code>v</code>
<code>v[1]</code>	→	acessa o segundo elemento de <code>v</code>
...		
<code>v[9]</code>	→	acessa o último elemento de <code>v</code>

Mas:

`v[10]` → está ERRADO (invasão de memória)

Para exemplificar o uso de vetores, vamos considerar um programa que lê 10 números reais, fornecidos via teclado, e calcula a média e a variância destes números. A média e a variância são dadas por:

$$m = \frac{\sum x}{N}, \quad v = \frac{\sum (x - m)^2}{N}$$

Uma possível implementação é apresentada a seguir.

```
/* Cálculo da media e da variância de 10 números reais */

#include <stdio.h>

int main ( void )
{
    float v[10];      /* declara vetor com 10 elementos */
    float med, var;   /* variáveis para armazenar a média e a variância */
    int i;             /* variável usada como índice do vetor */

    /* leitura dos valores */
    for (i = 0; i < 10; i++)           /* faz índice variar de 0 a 9 */
        scanf("%f", &v[i]);           /* lê cada elemento do vetor */

    /* cálculo da média */
    med = 0.0;                      /* inicializa média com zero */
    for (i = 0; i < 10; i++)
        med = med + v[i];           /* acumula soma dos elementos */
    med = med / 10;                 /* calcula a média */

    /* cálculo da variância */
    var = 0.0;                      /* inicializa variância com zero */
    for (i = 0; i < 10; i++)
        var = var+(v[i]-med)*(v[i]-med); /* acumula quadrado da diferença */
    var = var / 10;                 /* calcula a variância */

    printf ( "Media = %f      Variancia = %f\n", med, var );
    return 0;
}
```

Devemos observar que passamos para a função `scanf` o endereço de cada elemento do vetor (`&v[i]`), pois desejamos que os valores capturados sejam armazenados nos elementos do vetor. Se `v[i]` representa o $(i+1)$ -ésimo elemento do vetor, `&v[i]` representa o endereço de memória onde esse elemento está armazenado.

Na verdade, existe uma associação forte entre vetores e ponteiros, pois se existe a declaração:

```
int v[10];
```

a variável `v`, que representa o vetor, é uma constante que armazena o endereço inicial do vetor, isto é, `v`, sem indexação, aponta para o primeiro elemento do vetor.

A linguagem C também suporta aritmética de ponteiros. Podemos somar e subtrair ponteiros, desde que o valor do ponteiro resultante aponte para dentro da área reservada para o vetor. Se p representa um ponteiro para um inteiro, $p+1$ representa um ponteiro para o próximo inteiro armazenado na memória, isto é, o valor de p é incrementado de 4 (mais uma vez assumindo que um inteiro tem 4 bytes). Com isto, num vetor temos as seguintes equivalências:

$v+0$	\rightarrow	<i>aponta para o primeiro elemento do vetor</i>
$v+1$	\rightarrow	<i>aponta para o segundo elemento do vetor</i>
$v+2$	\rightarrow	<i>aponta para o terceiro elemento do vetor</i>
\dots		
$v+9$	\rightarrow	<i>aponta para o último elemento do vetor</i>

Portanto, escrever $\&v[i]$ é equivalente a escrever $(v+i)$. De maneira análoga, escrever $v[i]$ é equivalente a escrever $*(v+i)$ (é lógico que a forma indexada é mais clara e adequada). Devemos notar que o uso da aritmética de ponteiros aqui é perfeitamente válido, pois os elementos dos vetores são armazenados de forma contínua na memória.

Os vetores também podem ser inicializados na declaração:

```
int v[5] = { 5, 10, 15, 20, 25 };
```

ou simplesmente:

```
int v[] = { 5, 10, 15, 20, 25 };
```

Neste último caso, a linguagem dimensiona o vetor pelo número de elementos inicializados.

Passagem de vetores para funções

Passar um vetor para uma função consiste em passar o endereço da primeira posição do vetor. Se passarmos um valor de endereço, a função chamada deve ter um parâmetro do tipo ponteiro para armazenar este valor. Assim, se passarmos para uma função um vetor de `int`, devemos ter um parâmetro do tipo `int*`, capaz de armazenar endereços de inteiros. Salientamos que a expressão “passar um vetor para uma função” deve ser interpretada como “passar o endereço inicial do vetor”. Os elementos do vetor não são copiados para a função, o argumento copiado é apenas o endereço do primeiro elemento.

Para exemplificar, vamos modificar o código do exemplo acima, usando funções separadas para o cálculo da média e da variância. (Aqui, usamos ainda os operadores de atribuição `+=` para acumular as somas.)

```
/* Cálculo da media e da variância de 10 reais (segunda versão) */

#include <stdio.h>

/* Função para cálculo da média */
float media (int n, float* v)
{
    int i;
```

```

float s = 0.0;
for (i = 0; i < n; i++)
    s += v[i];
return s/n;
}

/* Função para cálculo da variância */
float variancia (int n, float* v, float m)
{
    int i;
    float s = 0.0;
    for (i = 0; i < n; i++)
        s += (v[i] - m) * (v[i] - m);
    return s/n;
}

int main ( void )
{
    float v[10];
    float med, var;
    int i;

    /* leitura dos valores */
    for ( i = 0; i < 10; i++ )
        scanf("%f", &v[i]);

    med = media(10,v);
    var = variancia(10,v,med);

    printf ( "Media = %f      Variância = %f\n", med, var );
    return 0;
}

```

Observamos ainda que, como é passado para a função o endereço do primeiro elemento do vetor (e não os elementos propriamente ditos), podemos alterar os valores dos elementos do vetor dentro da função. O exemplo abaixo ilustra:

```

/* Incrementa elementos de um vetor */

#include <stdio.h>

void incr_vetor ( int n, int *v )
{
    int i;
    for ( i = 0; i < n; i++)
        v[i]++;
}

int main ( void )
{
    int a[ ] = {1, 3, 5};
    incr_vetor(3, a);
    printf("%d %d %d\n", a[0], a[1], a[2]);
    return 0;
}

```

A saída do programa é 2 4 6, pois os elementos do vetor serão incrementados dentro da função.

5.2. Alocação dinâmica

Até aqui, na declaração de um vetor, foi preciso dimensioná-lo. Isto nos obrigava a saber, de antemão, quanto espaço seria necessário, isto é, tínhamos que prever o número máximo de elementos no vetor durante a codificação. Este pré-dimensionamento do vetor é um fator limitante. Por exemplo, se desenvolvermos um programa para calcular a média e a variância das notas de uma prova, teremos que prever o número máximo de alunos. Uma solução é dimensionar o vetor com um número absurdamente alto para não termos limitações quando da utilização do programa. No entanto, isto levaria a um desperdício de memória que é inaceitável em diversas aplicações. Se, por outro lado, formos modestos no pré-dimensionamento do vetor, o uso do programa fica muito limitado, pois não conseguiríamos tratar turmas com o número de alunos maior que o previsto.

Felizmente, a linguagem C oferece meios de requisitarmos espaços de memória em tempo de execução. Dizemos que podemos alocar memória dinamicamente. Com este recurso, nosso programa para o cálculo da média e variância discutido acima pode, em tempo de execução, consultar o número de alunos da turma e então fazer a alocação do vetor dinamicamente, sem desperdício de memória.

Uso da memória

Informalmente, podemos dizer que existem três maneiras de reservarmos espaço de memória para o armazenamento de informações. A primeira delas é através do uso de variáveis globais (e estáticas). O espaço reservado para uma variável global existe enquanto o programa estiver sendo executado. A segunda maneira é através do uso de variáveis locais. Neste caso, como já discutimos, o espaço existe apenas enquanto a função que declarou a variável está sendo executada, sendo liberado para outros usos quando a execução da função termina. Por este motivo, a função que chama não pode fazer referência ao espaço local da função chamada. As variáveis globais ou locais podem ser simples ou vetores. Para os vetores, precisamos informar o número máximo de elementos, caso contrário o compilador não saberia o tamanho do espaço a ser reservado.

A terceira maneira de reservarmos memória é requisitando ao sistema, em tempo de execução, um espaço de um determinado tamanho. Este espaço alocado dinamicamente permanece reservado até que explicitamente seja liberado pelo programa. Por isso, podemos alocar dinamicamente um espaço de memória numa função e acessá-lo em outra. A partir do momento que liberarmos o espaço, ele estará disponibilizado para outros usos e não podemos mais acessá-lo. Se o programa não liberar um espaço alocado, este será automaticamente liberado quando a execução do programa terminar.

Apresentamos abaixo um *esquema didático* que ilustra de maneira fictícia a distribuição do uso da memória pelo sistema operacional¹.

¹ A rigor, a alocação dos recursos é bem mais complexa e varia para cada sistema operacional.

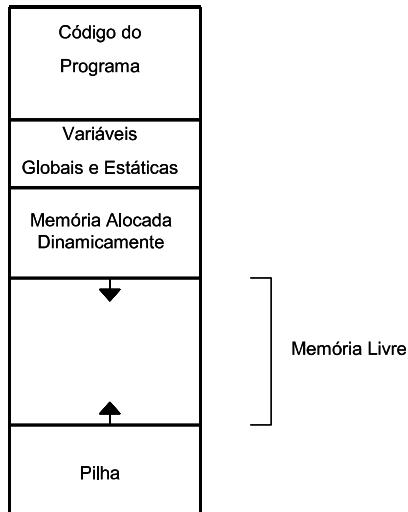


Figura 5.2: Alocação esquemática de memória.

Quando requisitamos ao sistema operacional para executar um determinado programa, o código em linguagem de máquina do programa deve ser carregado na memória, conforme discutido no primeiro capítulo. O sistema operacional reserva também os espaços necessários para armazenarmos as variáveis globais (e estáticas) existentes no programa. O restante da memória livre é utilizado pelas variáveis locais e pelas variáveis alocadas dinamicamente. Cada vez que uma determinada função é chamada, o sistema reserva o espaço necessário para as variáveis locais da função. Este espaço pertence à pilha de execução e, quando a função termina, é desempilhado. A parte da memória não ocupada pela pilha de execução pode ser requisitada dinamicamente. Se a pilha tentar crescer mais do que o espaço disponível existente, dizemos que ela “estourou” e o programa é abortado com erro. Similarmente, se o espaço de memória livre for menor que o espaço requisitado dinamicamente, a alocação não é feita e o programa pode prever um tratamento de erro adequado (por exemplo, podemos imprimir a mensagem “Memória insuficiente” e interromper a execução do programa).

Funções da biblioteca padrão

Existem funções, presentes na biblioteca padrão *stdlib*, que permitem alocar e liberar memória dinamicamente. A função básica para alocar memória é `malloc`. Ela recebe como parâmetro o número de bytes que se deseja alocar e retorna o endereço inicial da área de memória alocada.

Para exemplificar, vamos considerar a alocação dinâmica de um vetor de inteiros com 10 elementos. Como a função `malloc` retorna o endereço da área alocada e, neste exemplo, desejamos armazenar valores inteiros nessa área, devemos declarar um ponteiro de inteiro para receber o endereço inicial do espaço alocado. O trecho de código então seria:

```
int *v;
v = malloc(10*4);
```

Após este comando, se a alocação for bem sucedida, `v` armazenará o endereço inicial de uma área contínua de memória suficiente para armazenar 10 valores inteiros. Podemos,

então, tratar `v` como tratamos um vetor declarado estaticamente, pois, se `v` aponta para o inicio da área alocada, podemos dizer que `v[0]` acessa o espaço para o primeiro elemento que armazenaremos, `v[1]` acessa o segundo, e assim por diante (até `v[9]`).

No exemplo acima, consideramos que um inteiro ocupa 4 bytes. Para ficarmos independentes de compiladores e máquinas, usamos o operador `sizeof()`.

```
v = malloc(10*sizeof(int));
```

Além disso, devemos lembrar que a função `malloc` é usada para alocar espaço para armazenarmos valores de qualquer tipo. Por este motivo, `malloc` retorna um ponteiro genérico, para um tipo qualquer, representado por `void*`, que pode ser convertido automaticamente pela linguagem para o tipo apropriado na atribuição. No entanto, é comum fazermos a conversão explicitamente, utilizando o operador de molde de tipo (*cast*). O comando para a alocação do vetor de inteiros fica então:

```
v = (int *) malloc(10*sizeof(int));
```

A figura abaixo ilustra de maneira esquemática o que ocorre na memória:

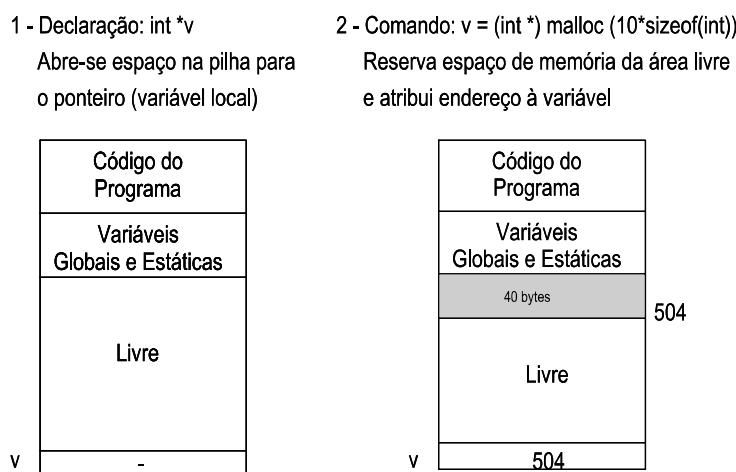


Figura 5.3: Alocação dinâmica de memória.

Se, porventura, não houver espaço livre suficiente para realizar a alocação, a função retorna um endereço nulo (representado pelo símbolo `NULL`, definido em `stdlib.h`). Podemos cercar o erro na alocação do programa verificando o valor de retorno da função `malloc`. Por exemplo, podemos imprimir uma mensagem e abortar o programa com a função `exit`, também definida na `stdlib`.

```

...
v = (int*) malloc(10*sizeof(int));
if (v==NULL)
{
    printf("Memoria insuficiente.\n");
    exit(1); /* aborta o programa e retorna 1 para o sist. operacional */
}
...

```

Para liberar um espaço de memória alocado dinamicamente, usamos a função `free`. Esta função recebe como parâmetro o ponteiro da memória a ser liberada. Assim, para liberar o vetor `v`, fazemos:

```
free (v);
```

Só podemos passar para a função `free` um endereço de memória que tenha sido alocado dinamicamente. Devemos lembrar ainda que não podemos acessar o espaço na memória depois que o liberamos.

Para exemplificar o uso da alocação dinâmica, alteraremos o programa para o cálculo da média e da variância mostrado anteriormente. Agora, o programa lê o número de valores que serão fornecidos, aloca um vetor dinamicamente e faz os cálculos. Somente a função principal precisa ser alterada, pois as funções para calcular a média e a variância anteriormente apresentadas independem do fato de o vetor ter sido alocado estática ou dinamicamente.

```

/* Cálculo da média e da variância de n reais */

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

...

int main ( void )
{
    int i, n;
    float *v;
    float med, var;

    /* leitura do número de valores */
    scanf("%d", &n);
    /* alocação dinâmica */
    v = (float*) malloc(n*sizeof(float));
    if (v==NULL) {
        printf("Memoria insuficiente.\n");
        return 1;
    }
    /* leitura dos valores */
    for (i = 0; i < n; i++)
        scanf("%f", &v[i]);
    med = media(n,v);
    var = variancia(n,v,med);
    printf("Media = %f      Variancia = %f\n", med, var);
    /* libera memória */
    free(v);
    return 0;
}
```

6. Cadeia de caracteres

W. Celes e J. L. Rangel

6.1. Caracteres

Efetivamente, a linguagem C não oferece um tipo caractere. Os caracteres são representados por códigos numéricos. A linguagem oferece o tipo `char`, que pode armazenar valores inteiros “pequenos”: um `char` tem tamanho de 1 byte, 8 bits, e sua versão com sinal pode representar valores que variam de -128 a 127. Como os códigos associados aos caracteres estão dentro desse intervalo, usamos o tipo `char` para representar caracteres¹. A correspondência entre os caracteres e seus códigos numéricos é feita por uma tabela de códigos. Em geral, usa-se a tabela ASCII, mas diferentes máquinas podem usar diferentes códigos. Contudo, se desejamos escrever códigos portáteis, isto é, que possam ser compilados e executados em máquinas diferentes, devemos evitar o uso explícito dos códigos referentes a uma determinada tabela, como será discutido nos exemplos subsequentes. Como ilustração, mostramos a seguir os códigos associados a alguns caracteres segundo a tabela ASCII.

Alguns caracteres que podem ser impressos (`sp` representa o branco, ou espaço):

	0	1	2	3	4	5	6	7	8	9
30			sp	!	"	#	\$	%	&	'
40	()	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[\]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~			

Alguns caracteres de controle:

0	nul	<i>null</i> : nulo
7	bel	<i>bell</i> : campainha
8	bs	<i>backspace</i> : voltar e apagar um caractere
9	ht	<i>tab</i> ou tabulação horizontal
10	nl	<i>newline</i> ou <i>line feed</i> : mudança de linha
13	cr	<i>carriage return</i> : volta ao início da linha
127	del	<i>delete</i> : apagar um caractere

¹ Alguns alfabetos precisam de maior representatividade. O alfabeto chinês, por exemplo, tem mais de 256 caracteres, não sendo suficiente o tipo `char` (alguns compiladores oferecem o tipo `wchar`, para estes casos).

Em C, a diferença entre caracteres e inteiros é feita apenas através da maneira pela qual são tratados. Por exemplo, podemos imprimir o mesmo valor de duas formas diferentes usando formatos diferentes. Vamos analisar o fragmento de código abaixo:

```
char c = 97;
printf("%d %c\n", c, c);
```

Considerando a codificação de caracteres via tabela ASCII, a variável `c`, que foi inicializada com o valor 97, representa o caractere `a`. A função `printf` imprime o conteúdo da variável `c` usando dois formatos distintos: com o especificador de formato para inteiro, `%d`, será impresso o valor do código numérico, 97; com o formato de caractere, `%c`, será impresso o caractere associado ao código, a letra `a`.

Conforme mencionamos, devemos evitar o uso explícito de códigos de caracteres. Para tanto, a linguagem C permite a escrita de *constantes caracteres*. Uma constante caractere é escrita envolvendo o caractere com aspas simples. Assim, a expressão '`a`' representa uma constante caractere e resulta no valor numérico associado ao caractere `a`. Podemos, então, reescrever o fragmento de código acima sem particularizar a tabela ASCII.

```
char c = 'a';
printf("%d %c\n", c, c);
```

Além de agregar portabilidade e clareza ao código, o uso de constantes caracteres nos livra de conhecermos os códigos associados a cada caractere.

Independentemente da tabela de códigos numéricos utilizada, garante-se que os dígitos são codificados em seqüência. Deste modo, se o dígito *zero* tem código 48, o dígito *um* tem obrigatoriamente código 49, e assim por diante. As letras minúsculas e as letras maiúsculas também formam dois grupos de códigos seqüenciais. O exemplo a seguir tira proveito desta seqüência dos códigos de caracteres.

Exemplo. Suponhamos que queremos escrever uma função para testar se um caractere `c` é um *dígito* (um dos caracteres entre '`0`' e '`9`'). Esta função pode ter o protótipo:

```
int digito(char c);
```

e ter como resultado 1 (verdadeiro) se `c` for um dígito, e 0 (falso) se não for.

A implementação desta função pode ser dada por:

```
int digito(char c)
{
    if ((c>='0') && (c<='9'))
        return 1;
    else
        return 0;
}
```

Exercício. Escreva uma função para determinar se um caractere é uma letra, com protótipo:

```
int letra(char c);
```

Exercício. Escreva uma função para converter um caractere para maiúscula. Se o caractere dado representar uma letra minúscula, devemos ter como valor de retorno a letra maiúscula correspondente. Se o caractere dado não for uma letra minúscula, devemos ter como valor de retorno o mesmo caractere, sem alteração. O protótipo desta função pode ser dado por:

```
char maiuscula(char c);
```

6.2. Cadeia de caracteres (*strings*)

Cadeias de caracteres (*strings*), em C, são representadas por vetores do tipo `char` terminadas, *obrigatoriamente*, pelo caractere nulo ('\0'). Portanto, para armazenarmos uma cadeia de caracteres, devemos reservar uma posição adicional para o caractere de fim da cadeia. Todas as funções que manipulam cadeias de caracteres (e a biblioteca padrão de C oferece várias delas) recebem como parâmetro um vetor de `char`, isto é, um ponteiro para o primeiro elemento do vetor que representa a cadeia, e processam caractere por caractere, até encontrarem o caractere nulo, que sinaliza o final da cadeia.

Por exemplo, o especificador de formato `%s` da função `printf` permite imprimir uma cadeia de caracteres. A função `printf` então recebe um vetor de `char` e imprime elemento por elemento, até encontrar o caractere nulo.

O código abaixo ilustra a representação de uma cadeia de caracteres. Como queremos representar a palavra `Rio`, composta por 3 caracteres, declaramos um vetor com dimensão 4 (um elemento adicional para armazenarmos o caractere nulo no final da cadeia). O código preenche os elementos do vetor, incluindo o caractere '\0', e imprime a palavra na tela.

```
int main ( void )
{
    char cidade[4];
    cidade[0] = 'R';
    cidade[1] = 'i';
    cidade[2] = 'o';
    cidade[3] = '\0';
    printf("%s \n", cidade);
    return 0;
}
```

Se o caractere '\0' não fosse colocado, a função `printf` executaria de forma errada, pois não conseguiria identificar o final da cadeia.

Como as cadeias de caracteres são vetores, podemos reescrever o código acima inicializando os valores dos elementos do vetor na declaração:

```
int main ( void )
```

```

{
    char cidade[ ] = {'R', 'i', 'o', '\0'};
    printf("%s \n", cidade);
    return 0;
}

```

A inicialização de cadeias de caracteres é tão comum em códigos C que a linguagem permite que elas sejam inicializadas escrevendo-se os caracteres entre aspas duplas. Neste caso, o caractere nulo é representado implicitamente. O código acima pode ser reescrito da seguinte forma:

```

int main ( void )
{
    char cidade[ ] = "Rio";
    printf("%s \n", cidade);
    return 0;
}

```

A variável `cidade` é automaticamente dimensionada e inicializada com 4 elementos. Para ilustrar a declaração e inicialização de cadeias de caracteres, consideremos as declarações abaixo:

```

char s1[] = "";
char s2[] = "Rio de Janeiro";
char s3[81];
char s4[81] = "Rio";

```

Nestas declarações, a variável `s1` armazena uma cadeia de caracteres vazia, representada por um vetor com um único elemento, o caractere '`\0`'. A variável `s2` representa um vetor com 15 elementos. A variável `s3` representa uma cadeia de caracteres capaz de representar cadeias com até 80 caracteres, já que foi dimensionada com 81 elementos. Esta variável, no entanto, não foi inicializada e seu conteúdo é desconhecido. A variável `s4` também foi dimensionada para armazenar cadeias até 80 caracteres, mas seus primeiros quatro elementos foram atribuídos na declaração.

Leitura de caracteres e cadeias de caracteres

Para capturarmos o valor de um caractere simples fornecido pelo usuário via teclado, usamos a função `scanf`, com o especificador de formato `%c`.

```

char a;
...
scanf("%c", &a);
...

```

Desta forma, se o usuário digitar a letra `r`, por exemplo, o código associado à letra `r` será armazenado na variável `a`. Vale ressaltar que, diferente dos especificadores `%d` e `%f`, o especificador `%c` não pula os caracteres brancos². Portanto, se o usuário teclar um espaço

² Um “caractere branco” pode ser um espaço (' '), um caractere de tabulação ('\t') ou um caractere de nova linha ('\n').

antes da letra `r`, o código do espaço será capturado e a letra `r` será capturada apenas numa próxima chamada da função `scanf`. Se desejarmos pular todas as ocorrências de caracteres brancos que porventura antecedam o caractere que queremos capturar, basta incluir um espaço em branco no formato, antes do especificador.

```
char a;
...
scanf(" %c", &a); /* o branco no formato pula brancos da entrada */
...
```

Já mencionamos que o especificador `%s` pode ser usado na função `printf` para imprimir uma cadeia de caracteres. O mesmo especificador pode ser utilizado para capturar cadeias de caracteres na função `scanf`. No entanto, seu uso é muito limitado. O especificador `%s` na função `scanf` pula os eventuais caracteres brancos e capture a seqüência de caracteres não brancos. Consideremos o fragmento de código abaixo:

```
char cidade[81];
...
scanf("%s", cidade);
...
```

Devemos notar que não usamos o caractere `&` na passagem da cadeia para a função, pois a cadeia é um vetor (o nome da variável representa o endereço do primeiro elemento do vetor e a função atribui os valores dos elementos a partir desse endereço). O uso do especificador de formato `%s` na leitura é limitado, pois o fragmento de código acima funciona apenas para capturar nomes simples. Se o usuário digitar `Rio de Janeiro`, apenas a palavra `Rio` será capturada, pois o `%s` lê somente uma seqüência de caracteres não brancos.

Em geral, queremos ler nomes compostos (nome de pessoas, cidades, endereços para correspondência, etc.). Para capturarmos estes nomes, podemos usar o especificador de formato `%[...]`, no qual listamos entre os colchetes todos os caracteres que aceitaremos na leitura. Assim, o formato `"%[aeiou]"` lê seqüências de vogais, isto é, a leitura prossegue até que se encontre um caractere que não seja uma vogal. Se o primeiro caractere entre colchetes for o acento circunflexo (`^`), teremos o efeito inverso (negação). Assim, com o formato `"%^[aeiou]"` a leitura prossegue enquanto uma vogal não for encontrada. Esta construção permite capturarmos nomes compostos. Consideremos o código abaixo:

```
char cidade[81];
...
scanf(" %[^\\n]", cidade);
...
```

A função `scanf` agora lê uma seqüência de caracteres até que seja encontrado o caractere de mudança de linha (`'\\n'`). Em termos práticos, capture-se a linha fornecida pelo usuário até que ele tecle “*Enter*”. A inclusão do espaço no formato (antes do sinal `%`) garante que eventuais caracteres brancos que precedam o nome serão pulados.

Para finalizar, devemos salientar que o trecho de código acima é perigoso, pois, se o usuário fornecer uma linha que tenha mais de 80 caracteres, estaremos invadindo um espaço de memória que não está reservado (o vetor foi dimensionado com 81 elementos).

Para evitar esta possível invasão, podemos limitar o número máximo de caracteres que serão capturados.

```
char cidade[81];
...
scanf(" %80[^\\n]", cidade);      /* lê no máximo 80 caracteres */
...
```

Exemplos de funções que manipulam cadeias de caracteres

Nesta seção, discutiremos a implementação de algumas funções que manipulam cadeias de caracteres.

Exemplo. Impressão caractere por caractere.

Vamos inicialmente considerar a implementação de uma função que imprime uma cadeia de caracteres, caractere por caractere. A implementação pode ser dada por:

```
void imprime (char* s)
{
    int i;
    for (i=0; s[i] != '\0'; i++)
        printf("%c", s[i]);
    printf("\n");
}
```

que teria funcionalidade análoga à utilização do especificador de formato `%s`.

```
void imprime (char* s)
{
    printf("%s\n", s);
}
```

Exemplo. Comprimento da cadeia de caracteres.

Consideremos a implementação de uma função que recebe como parâmetro de entrada uma cadeia de caracteres e fornece como retorno o número de caracteres existentes na cadeia. O protótipo da função pode ser dado por:

```
int comprimento (char* s);
```

Para contar o número de caracteres da cadeia, basta contarmos o número de caracteres até que o caractere nulo (que indica o fim da cadeia) seja encontrado. O caractere nulo em si não deve ser contado. Uma possível implementação desta função é:

```
int comprimento (char* s)
{
    int i;
    int n = 0; /* contador */
    for (i=0; s[i] != '\0'; i++)
        n++;
    return n;
}
```

O trecho de código abaixo faz uso da função acima.

```
#include <stdio.h>

int comprimento (char* s);

int main (void)
{
    int tam;
    char cidade[] = "Rio de Janeiro";
    tam = comprimento(cidade);
    printf("A string \"%s\" tem %d caracteres\n", cidade, tam);
    return 0;
}
```

A saída deste programa será: A string "Rio de Janeiro" tem 14 caracteres. Salientamos o uso do caractere de escape \" para incluir as aspas na saída.

Exemplo. Cópia de cadeia de caracteres.

Vamos agora considerar a implementação de uma função para copiar os elementos de uma cadeia de caracteres para outra. Assumimos que a cadeia que receberá a cópia tem espaço suficiente para que a operação seja realizada. O protótipo desta função pode ser dado por:

```
void copia (char* dest, char* orig);
```

A função copia os elementos da cadeia original (*orig*) para a cadeia de destino (*dest*). Uma possível implementação desta função é mostrada abaixo:

```
void copia (char* dest, char* orig)
{
    int i;
    for (i=0; orig[i] != '\0'; i++)
        dest[i] = orig[i];
    /* fecha a cadeia copiada */
    dest[i] = '\0';
}
```

Salientamos a necessidade de “fechar” a cadeia copiada após a cópia dos caracteres não nulos. Quando o laço do *for* terminar, a variável *i* terá o índice de onde está armazenado o caractere nulo na cadeia original. A cópia também deve conter o '\0' nesta posição.

Exemplo. Concatenação de cadeias de caracteres.

Vamos considerar uma extensão do exemplo anterior e discutir a implementação de uma função para concatenar uma cadeia de caracteres com outra já existente. Isto é, os caracteres de uma cadeia são copiados no final da outra cadeia. Assim, se uma cadeia representa inicialmente a cadeia PUC e concatenarmos a ela a cadeia Rio, teremos como resultado a cadeia PUCRio. Vamos mais uma vez considerar que existe espaço reservado que permite fazer a cópia dos caracteres. O protótipo da função pode ser dado por:

```
void concatena (char*dest, char* orig);
```

Uma possível implementação desta função é mostrada a seguir:

```
void concatena (char*dest, char* orig)
{
    int i = 0; /* indice usado na cadeia destino, inicializado com zero */
    int j;      /* indice usado na cadeia origem */
    /* acha o final da cadeia destino */
    i = 0;
    while (s[i] != '\0')
        i++;
    /* copia elementos da origem para o final do destino */
    for (j=0; orig[j] != '\0'; j++)
    {
        dest[i] = orig[j];
        i++;
    }
    /* fecha cadeia destino */
    dest[i] = '\0';
}
```

Funções análogas às funções `comprimento`, `copia` e `concatena` são disponibilizadas pela biblioteca padrão de C. As funções da biblioteca padrão são, respectivamente, `strlen`, `strcpy` e `strcat`, que fazem parte da biblioteca de cadeias de caracteres (*strings*), `string.h`. Existem diversas outras funções que manipulam cadeias de caracteres nessa biblioteca. A razão de mostrarmos possíveis implementações destas funções como exercício é ilustrar a codificação da manipulação de cadeias de caracteres.

Exemplo 5: Duplicação de cadeias de caracteres.

Consideremos agora um exemplo com alocação dinâmica. O objetivo é implementar uma função que receba como parâmetro uma cadeia de caracteres e forneça uma cópia da cadeia, alocada dinamicamente. O protótipo desta função pode ser dado por:

```
char* duplica (char* s);
```

Uma possível implementação, usando as funções da biblioteca padrão, é:

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

char* duplica (char* s)
{
    int n = strlen(s);
    char* d = (char*) malloc ((n+1)*sizeof(char));
    strcpy(d,s);
    return d;
}
```

A função que chama `duplica` fica responsável por liberar o espaço alocado.

Funções recursivas

Uma cadeia de caracteres pode ser definida de forma recursiva. Podemos dizer que uma cadeia de caracteres é representada por:

- uma cadeia de caracteres vazia; ou
- um caractere seguido de uma (sub) cadeia de caracteres.

Isto é, podemos dizer que uma cadeia s não vazia pode ser representada pelo seu primeiro caractere $s[0]$ seguido da cadeia que começa no endereço do então segundo caractere, $\&s[1]$.

Vamos reescrever algumas das funções mostradas acima, agora com a versão recursiva.

Exemplo. Impressão caractere por caractere.

Uma versão recursiva da função para imprimir a cadeia caractere por caractere é mostrada a seguir. Como já foi discutido, uma implementação recursiva deve ser projetada considerando-se a definição recursiva do objeto, no caso uma cadeia de caracteres. Assim, a função deve primeiro testar se a condição da cadeia é vazia. Se a cadeia for vazia, nada precisa ser impresso; se não for vazia, devemos imprimir o primeiro caractere e então chamar uma função para imprimir a sub-cadeia que se segue. Para imprimir a sub-cadeia podemos usar a própria função, recursivamente.

```
void imprime_rec (char* s)
{
    if (s[0] != '\0')
    {
        printf("%c", s[0]);
        imprime_rec(&s[1]);
    }
}
```

Algumas implementações ficam bem mais simples se feitas recursivamente. Por exemplo, é simples alterar a função acima e fazer com que os caracteres da cadeia sejam impressos em ordem inversa, de trás para a frente: basta imprimir a sub-cadeia antes de imprimir o primeiro caractere.

```
void imprime_inv (char* s)
{
    if (s[0] != '\0')
    {
        imprime_inv(&s[1]);
        printf("%c", s[0]);
    }
}
```

Como exercício, sugerimos implementar a impressão inversa sem usar recursividade.

Exemplo. Comprimento da cadeia de caracteres.

Uma implementação recursiva da função que retorna o número de caracteres existentes na cadeia é mostrada a seguir:

```
int comprimento_rec (char* s)
{
    if (s[0] == '\0')
        return 0;
    else
        return 1 + comprimento_rec(&s[1]);
}
```

Exemplo. Cópia de cadeia de caracteres.

Vamos mostrar agora uma possível implementação recursiva da função `copia` mostrada anteriormente.

```
void copia_rec (char* dest, char* orig)
{
    if (orig[0] == '\0')
        dest[0] = '\0';
    else {
        dest[0] = orig[0];
        copia_rec(&dest[1], &orig[1]);
    }
}
```

É fácil verificar que o código acima pode ser escrito de forma mais compacta:

```
void copia_rec_2 (char* dest, char* orig)
{
    dest[0] = orig[0];
    if (orig[0] != '\0')
        copia_rec_2(&dest[1], &orig[1]);
}
```

Constante cadeia de caracteres**

Em códigos C, uma seqüência de caracteres delimitada por aspas representa uma constante cadeia de caracteres, ou seja, uma expressão constante, cuja avaliação resulta no ponteiro onde a cadeia de caracteres está armazenada. Para exemplificar, vamos considerar o trecho de código abaixo:

```
#include <string.h>

int main ( void )
{
    char cidade[4];
    strcpy (cidade, "Rio");
    printf ( "%s \n", cidade );
    return 0;
}
```

De forma ilustrativa, o que acontece é que, quando o compilador encontra a cadeia "Rio", automaticamente é alocada na área de constantes a seguinte seqüência de caracteres:

```
'R', 'i', 'o', '\0'
```

e é fornecido o ponteiro para o primeiro elemento desta seqüência. Assim, a função `strcpy` recebe dois ponteiros de cadeias: o primeiro aponta para o espaço associado à variável `cidade` e o segundo aponta para a área de constantes onde está armazenada a cadeia `Rio`.

Desta forma, também é válido escrever:

```
int main (void)
{
    char *cidade; /* declara um ponteiro para char */
    cidade = "Rio"; /* cidade recebe o endereço da cadeia "Rio" */
    printf ("%s \n", cidade );
    return 0;
}
```

Existe uma diferença sutil entre as duas declarações abaixo:

```
char s1[] = "Rio de Janeiro";
char* s2 = "Rio de Janeiro";
```

Na primeira, declaramos um vetor de `char` local que é inicializado com a cadeia de caracteres Rio de Janeiro, seguido do caractere nulo. A variável `s1` ocupa, portanto, 15 bytes de memória. Na segunda, declaramos um ponteiro para `char` que é inicializado com o endereço de uma área de memória onde a constante cadeia de caracteres Rio de Janeiro está armazenada. A variável `s2` ocupa 4 bytes (espaço de um ponteiro). Podemos verificar esta diferença imprimindo os valores `sizeof(s1)` e `sizeof(s2)`. Como `s1` é um vetor local, podemos alterar o valor de seus elementos. Por exemplo, é válido escrever `s1[0]='X'`; alterando o conteúdo da cadeia para Xio de Janeiro. No entanto, não é válido escrever `s2[0]='X'`; pois estariamos tentando alterar o conteúdo de uma área de constante.

6.3. Vtor de cadeia de caracteres

Em muitas aplicações, desejamos representar um vetor de cadeia de caracteres. Por exemplo, podemos considerar uma aplicação que armazene os nomes de todos os alunos de uma turma num vetor. Sabemos que uma cadeia de caracteres é representada por um vetor do tipo `char`. Para representarmos um vetor onde cada elemento é uma cadeia de caracteres, devemos ter um vetor cujos elementos são do tipo `char*`, isto é, um vetor de ponteiros para `char`. Assim, criamos um conjunto (vetor) bidimensional de `char`. Assumindo que o nome de nenhum aluno terá mais do que 80 caracteres e que o número máximo de alunos numa turma é 50, podemos declarar um vetor bidimensional para armazenar os nomes dos alunos

```
char alunos[50][81];
```

Com esta variável declarada, `alunos[i]` acessa a cadeia de caracteres com o nome do $(i+1)$ -ésimo aluno da turma e, consequentemente, `alunos[i][j]` acessa a $(j+1)$ -ésima letra do nome do $(i+1)$ -ésimo aluno. Considerando que `alunos` é uma variável global, uma função para imprimir os nomes dos `n` alunos de uma turma poderia ser dada por:

```
void imprime (int n)
{
    int i;
    for (i=0; i<n; i++)
        printf("%s\n", alunos[i]);
}
```

No próximo capítulo, que trata de matrizes, discutiremos conjuntos bidimensionais com mais detalhes. Para a representação de vetores de cadeias de caracteres, optamos, em geral, por declarar um vetor de ponteiros e alocar dinamicamente cada elemento (no caso, uma cadeia de caracteres). Desta forma, otimizamos o uso do espaço de memória, pois não precisamos achar uma dimensão máxima para todas as cadeias do vetor nem desperdiçamos espaço excessivo quando temos poucos nomes de alunos a serem armazenados. Cada elemento do vetor é um ponteiro. Se precisarmos armazenar um nome na posição, alocamos o espaço de memória necessário para armazenar a cadeia de caracteres correspondente. Assim, nosso vetor com os nomes dos alunos pode ser declarado da seguinte forma:

```
#define MAX 50
char* alunos[MAX];
```

Exemplo. Leitura e impressão dos nomes dos alunos.

Vamos escrever uma função que captura os nomes dos alunos de uma turma. A função inicialmente lê o número de alunos da turma (que deve ser menor ou igual a MAX) e captura os nomes fornecidos por linha, fazendo a alocação correspondente. Para escrever esta função, podemos pensar numa função auxiliar que captura uma linha e fornece como retorno uma cadeia alocada dinamicamente com a linha inserida. Fazendo uso das funções que escrevemos acima, podemos ter:

```
char* lelinha (void)
{
    char linha[121]; /* variavel auxiliar para ler linha */
    scanf(" %120[^\\n]",linha);
    return duplica(linha);
}
```

A função para capturar os nomes dos alunos preenche o vetor de nomes e pode ter como valor de retorno o número de nomes lidos:

```
int lenomes (char** alunos)
{
    int i;
    int n;
    do {
        scanf("%d",&n);
    } while (n>MAX);

    for (i=0; i<n; i++)
        alunos[i] = lelinha();
    return n;
}
```

A função para liberar os nomes alocados na tabela pode ser implementada por:

```
void liberanomes (int n, char** alunos)
{
    int i;
    for (i=0; i<n; i++)
        free(alunos[i]);
}
```

Uma função para imprimir os nomes dos alunos pode ser dada por:

```
void imprimenomes (int n, char** alunos)
{
    int i;
    for (i=0; i<n; i++)
        printf("%s\n", alunos[i]);
}
```

Um programa que faz uso destas funções é mostrado a seguir:

```
int main (void)
{
    char* alunos[MAX];
    int n = lenomes(alunos);
    imprimenomes(n, alunos);
    liberanomes(n, alunos);
    return 0;
}
```

Parâmetros da função main**

Em todos os exemplos mostrados, temos considerado que a função principal, `main`, não recebe parâmetros. Na verdade, a função `main` pode ser definida para receber zero ou dois parâmetros, geralmente chamados `argc` e `argv`. O parâmetro `argc` recebe o número de argumentos passados para o programa quando este é executado; por exemplo, de um comando de linha do sistema operacional. O parâmetro `argv` é um vetor de cadeias de caracteres, que armazena os nomes passados como argumentos. Por exemplo, se temos um programa executável com o nome `mensagem` e se ele for invocado através da linha de comando:

```
> mensagem estruturas de dados
```

a variável `argc` receberá o valor 4 e o vetor `argv` será inicializado com os seguintes elementos: `argv[0] = "mensagem"`, `argv[1] = "estruturas"`, `argv[2] = "de"`, e `argv[3] = "dados"`. Isto é, o primeiro elemento armazena o próprio nome do executável e os demais são preenchidos com os nomes passados na linha de comando. Esses parâmetros podem ser úteis para, por exemplo, passar o nome de um arquivo de onde serão capturados os dados de um programa. A manipulação de arquivos será discutida mais adiante no curso. Por ora, mostramos um exemplo simples que trata os dois parâmetros da função `main`.

```
#include <stdio.h>
int main (int argc, char** argv)
{
    int i;
    for (i=0; i<argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

Se este programa tiver seu executável chamado de `mensagem` e for invocado com a linha de comando mostrada acima, a saída será:

```
mensagem
estruturas
de
dados
```

7. Tipos estruturados

W. Celes e J. L. Rangel

Na linguagem C, existem os tipos básicos (`char`, `int`, `float`, etc.) e seus respectivos ponteiros que podem ser usados na declaração de variáveis. Para estruturar dados complexos, nos quais as informações são compostas por diversos campos, necessitamos de mecanismos que nos permitam agrupar tipos distintos. Neste capítulo, apresentaremos os mecanismos fundamentais da linguagem C para a estruturação de tipos.

7.1. O tipo estrutura

Em C, podemos definir um tipo de dado cujos campos são compostos de vários valores de tipos mais simples. Para ilustrar, vamos considerar o desenvolvimento de programas que manipulam pontos no plano cartesiano. Cada ponto pode ser representado por suas coordenadas x e y , ambas dadas por valores reais. Sem um mecanismo para agrupar as duas componentes, teríamos que representar cada ponto por duas variáveis independentes.

```
float x;  
float y;
```

No entanto, deste modo, os dois valores ficam dissociados e, no caso do programa manipular vários pontos, cabe ao programador não misturar a coordenada x de um ponto com a coordenada y de outro. Para facilitar este trabalho, a linguagem C oferece recursos para agruparmos dados. Uma estrutura, em C, serve basicamente para agrupar diversas variáveis dentro de um único contexto. No nosso exemplo, podemos definir uma estrutura ponto que contenha as duas variáveis. A sintaxe para a definição de uma estrutura é mostrada abaixo:

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

Desta forma, a estrutura `ponto` passa a ser um tipo e podemos então declarar variáveis deste tipo.

```
struct ponto p;
```

Esta linha de código declara `p` como sendo uma variável do tipo `struct ponto`. Os elementos de uma estrutura podem ser acessados através do operador de acesso “ponto” (`.`). Assim, é válido escrever:

```
ponto.x = 10.0;  
ponto.y = 5.0;
```

Manipulamos os elementos de uma estrutura da mesma forma que variáveis simples. Podemos acessar seus valores, atribuir-lhes novos valores, acessar seus endereços, etc.

Exemplo: Capturar e imprimir as coordenadas de um ponto.

Para exemplificar o uso de estruturas em programas, vamos considerar um exemplo simples em que capturamos e imprimimos as coordenadas de um ponto qualquer.

```
/* Captura e imprime as coordenadas de um ponto qualquer */

#include <stdio.h>

struct ponto {
    float x;
    float y;
};

int main (void)
{
    struct ponto p;

    printf("Digite as coordenadas do ponto(x y): ");
    scanf("%f %f", &p.x, &p.y);
    printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
    return 0;
}
```

A variável `p`, definida dentro de `main`, é uma variável local como outra qualquer. Quando a declaração é encontrada, aloca-se, na pilha de execução, um espaço para seu armazenamento, isto é, um espaço suficiente para armazenar todos os campos da estrutura (no caso, dois números reais). Notamos que o acesso ao endereço de um campo da estrutura é feito da mesma forma que com variáveis simples: basta escrever `&(p.x)`, ou simplesmente `&p.x`, pois o operador de acesso ao campo da estrutura tem precedência sobre o operador “endereço de”.

Ponteiro para estruturas

Da mesma forma que podemos declarar variáveis do tipo estrutura:

```
struct ponto p;
```

podemos também declarar variáveis do tipo ponteiro para estrutura:

```
struct ponto *pp;
```

Se a variável `pp` armazenar o endereço de uma estrutura, podemos acessar os campos dessa estrutura indiretamente, através de seu ponteiro:

```
(*pp).x = 12.0;
```

Neste caso, os parênteses são indispensáveis, pois o operador “conteúdo de” tem precedência menor que o operador de acesso. O acesso de campos de estruturas é tão comum em programas C que a linguagem oferece outro operador de acesso, que permite acessar campos a partir do ponteiro da estrutura. Este operador é composto por um traço seguido de um sinal de maior, formando uma seta (`->`). Portanto, podemos reescrever a atribuição anterior fazendo:

```
pp->x = 12.0;
```

Em resumo, se temos uma variável estrutura e queremos acessar seus campos, usamos o operador de acesso ponto (`p.x`); se temos uma variável ponteiro para estrutura, usamos o operador de acesso seta (`pp->x`). Seguindo o raciocínio, se temos o ponteiro e queremos acessar o endereço de um campo, fazemos `&pp->x`!

Passagem de estruturas para funções

Para exemplificar a passagem de variáveis do tipo estrutura para funções, podemos reescrever o programa simples, mostrado anteriormente, que captura e imprime as coordenadas de um ponto qualquer. Inicialmente, podemos pensar em escrever uma função que imprima as coordenadas do ponto. Esta função poderia ser dada por:

```
void imprime (struct ponto p)
{
    printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
}
```

A passagem de estruturas para funções se processa de forma análoga à passagem de variáveis simples, porém exige uma análise mais detalhada. Da forma como está escrita no código acima, a função recebe uma estrutura inteira como parâmetro. Portanto, faz-se uma cópia de toda a estrutura para a pilha e a função acessa os dados desta cópia. Existem dois pontos a serem ressaltados. Primeiro, como em toda passagem por valor, a função não tem como alterar os valores dos elementos da estrutura original (na função `imprime` isso realmente não é necessário, mas seria numa função de leitura). O segundo ponto diz respeito à eficiência, visto que copiar uma estrutura inteira para a pilha pode ser uma operação custosa (principalmente se a estrutura for muito grande). É mais conveniente passar apenas o ponteiro da estrutura, mesmo que não seja necessário alterar os valores dos elementos dentro da função, pois copiar um ponteiro para a pilha é muito mais eficiente do que copiar uma estrutura inteira. Um ponteiro ocupa em geral 4 bytes, enquanto uma estrutura pode ser definida com um tamanho muito grande. Desta forma, uma segunda (e mais adequada) alternativa para escrevermos a função `imprime` é:

```
void imprime (struct ponto* pp)
{
    printf("O ponto fornecido foi: (%.2f,%.2f)\n", pp->x, pp->y);
}
```

Podemos ainda pensar numa função para ler a hora do evento. Observamos que, neste caso, obrigatoriamente devemos passar o ponteiro da estrutura, caso contrário não seria possível passar ao programa principal os dados lidos:

```
void captura (struct ponto* pp)
{
    printf("Digite as coordenadas do ponto(x y): ");
    scanf("%f %f", &p->x, &p->y);
}
```

Com estas funções, nossa função `main` ficaria como mostrado abaixo.

```
int main (void)
{
    struct ponto p;

    captura(&p);
    imprime(&p);

    return 0;
}
```

Exercício: Função para determinar a distância entre dois pontos.

Considere a implementação de uma função que tenha como valor de retorno a distância entre dois pontos. O protótipo da função pode ser dado por:

```
float distancia (struct ponto *p, struct ponto *q);
```

Nota: A distância entre dois pontos é dada por: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Alocação dinâmica de estruturas

Da mesma forma que os vetores, as estruturas podem ser alocadas dinamicamente. Por exemplo, é válido escrever:

```
struct ponto* p;
p = (struct ponto*) malloc (sizeof(struct ponto));
```

Neste fragmento de código, o tamanho do espaço de memória alocado dinamicamente é dado pelo operador `sizeof` aplicado sobre o tipo estrutura (`sizeof(struct ponto)`). A função `malloc` retorna o endereço do espaço alocado, que é então convertido para o tipo ponteiro da estrutura `ponto`.

Após uma alocação dinâmica, podemos acessar normalmente os campos da estrutura, através da variável ponteiro que armazena seu endereço:

```
...
p->x = 12.0;
...
```

7.2. Definição de "novos" tipos

A linguagem C permite criar nomes de tipos. Por exemplo, se escrevermos:

```
typedef float Real;
```

podemos usar o nome `Real` como um mnemônico para o tipo `float`. O uso de `typedef` é muito útil para abreviarmos nomes de tipos e para tratarmos tipos complexos. Alguns exemplos válidos de `typedef`:

```
typedef unsigned char UChar;
typedef int* PInt;
typedef float Vetor[4];
```

Neste fragmento de código, definimos `UChar` como sendo o tipo `char` sem sinal, `PInt` como um tipo ponteiro para `int`, e `Vetor` como um tipo que representa um vetor de quatro elementos. A partir dessas definições, podemos declarar variáveis usando estes mnemônicos:

```
Vetor v;
...
v[0] = 3;
...
```

Em geral, definimos nomes de tipos para as estruturas com as quais nossos programas trabalham. Por exemplo, podemos escrever:

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

typedef struct ponto Ponto;
```

Neste caso, `Ponto` passa a representar nossa estrutura de ponto. Também podemos definir um nome para o tipo ponteiro para a estrutura.

```
typedef struct ponto *PPonto;
```

Podemos ainda definir mais de um nome num mesmo `typedef`. Os dois `typedef` anteriores poderiam ser escritos por:

```
typedef struct ponto Ponto, *PPonto;
```

A sintaxe de um `typedef` pode parecer confusa, mas é equivalente à da declaração de variáveis. Por exemplo, na definição abaixo:

```
typedef float Vector[4];
```

se omitíssemos a palavra `typedef`, estaríamos declarando a variável `Vector` como sendo um vetor de 4 elementos do tipo `float`. Com `typedef`, estamos definindo um nome que representa o tipo vetor de 4 elementos `float`. De maneira análoga, na definição:

```
typedef struct ponto Ponto, *PPonto;
```

se omitíssemos a palavra `typedef`, estaríamos declarando a variável `Ponto` como sendo do tipo `struct ponto` e a variável `PPonto` como sendo do tipo ponteiro para `struct ponto`.

Por fim, vale salientar que podemos definir a estrutura e associar mnemônicos para elas em um mesmo comando:

```
typedef struct ponto {
    float x;
    float y;
} Ponto, *PPonto;
```

É comum os programadores de C usarem nomes com as primeiras letras maiúsculas na definição de tipos. Isso não é uma obrigatoriedade, apenas um estilo de codificação.

7.3. Vetores de estruturas

Já discutimos o uso de vetores para agrupar elementos dos tipos básicos (vetores de inteiros, por exemplo). Nesta seção, vamos discutir o uso de vetores de estruturas, isto é, vetores cujos elementos são estruturas. Para ilustrar a discussão, vamos considerar o cálculo da área de um polígono plano qualquer delimitado por uma seqüência de n pontos. A área desse polígono pode ser calculada somando-se as áreas dos trapézios formados pelos lados do polígono e o eixo x , conforme ilustra a Figura 7.1.

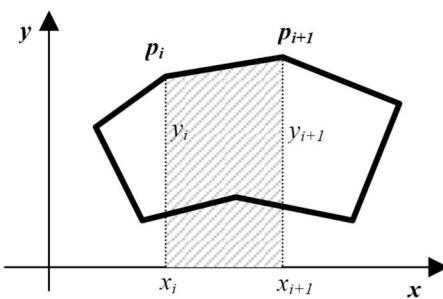


Figura 7.1: Cálculo da área de um polígono.

Na figura, ressaltamos a área do trapézio definido pela aresta que vai do ponto p_i ao ponto p_{i+1} . A área desse trapézio é dada por: $a = (x_{i+1} - x_i)(y_{i+1} + y_i)/2$. Somando-se as “áreas” (algumas delas negativas) dos trapézios definidos por todas as arestas chega-se a área do polígono (as áreas externas ao polígono são anuladas). Se a seqüência de pontos que define o polígono for dada em sentido anti-horário, chega-se a uma “área” de valor negativo. Neste caso, a área do polígono é o valor absoluto do resultado da soma.

Um vetor de estruturas pode ser usado para definir um polígono. O polígono passa a ser representado por um seqüência de pontos. Podemos, então, escrever uma função para calcular a área de um polígono, dados o número de pontos e o vetor de pontos que o representa. Uma implementação dessa função é mostrada abaixo.

```
float area (int n, Ponto* p)
{
    int i, j;
    float a = 0;
    for (i=0; i<n; i++) {
        j = (i+1) % n; /* próximo índice (incremento circular) */
        a += (p[j].x-p[i].x)*(p[i].y + p[j].y)/2;
    }

    if (a < 0)
        return -a;
    else
        return a;
}
```

Um exemplo de uso dessa função é mostrado no código abaixo:

```
int main (void)
{
    Ponto p[3] = {{1.0,1.0},{5.0,1.0},{4.0,3.0}};
    printf("area = %f\n",area (3,p));
    return 0;
}
```

Exercício: Altere o programa acima para capturar do teclado o número de pontos que delimitam o polígono. O programa deve, então, alocar dinamicamente o vetor de pontos, capturar as coordenadas dos pontos e, chamando a função `area`, exibir o valor da área.

7.4. *Vetores de ponteiros para estruturas*

Da mesma forma que podemos declarar vetores de estruturas, podemos também declarar vetores de ponteiros para estruturas. O uso de vetores de ponteiros é útil quando temos que tratar um conjunto elementos complexos. Para ilustrar o uso de estruturas complexas, consideremos um exemplo em que desejamos armazenar uma tabela com dados de alunos. Podemos organizar os dados dos alunos em um vetor. Para cada aluno, vamos supor que sejam necessárias as seguintes informações:

- *nome*: cadeia com até 80 caracteres
- *matricula*: número inteiro
- *endereço*: cadeia com até 120 caracteres
- *telefone*: cadeia com até 20 caracteres

Para estruturar esses dados, podemos definir um tipo que representa os dados de um aluno:

```
struct aluno {
    char nome[81];
    int mat;
    char end[121];
    char tel[21];
};

typedef struct aluno Aluno;
```

Vamos montar a tabela de alunos usando um vetor global com um número máximo de alunos. Uma primeira opção é declarar um vetor de estruturas:

```
#define MAX 100
Aluno tab[MAX];
```

Desta forma, podemos armazenar nos elementos do vetor os dados dos alunos que queremos organizar. Seria válido, por exemplo, uma atribuição do tipo:

```
...
tab[i].mat = 9912222;
...
```

No entanto, o uso de vetores de estruturas tem, neste caso, uma grande desvantagem. O tipo `Aluno` definido acima ocupa pelo menos 227 ($=81+4+121+21$) bytes¹. A declaração de um vetor desta estrutura representa um desperdício significativo de memória, pois provavelmente estaremos armazenando de fato um número de alunos bem inferior ao máximo estimado. Para contornar este problema, podemos trabalhar com um vetor de ponteiros.

```
typedef struct aluno *PAluno;

#define MAX 100
PAluno tab[MAX];
```

Assim, cada elemento do vetor ocupa apenas o espaço necessário para armazenar um ponteiro. Quando precisarmos alocar os dados de um aluno numa determinada posição do vetor, alocamos dinamicamente a estrutura `Aluno` e guardamos seu endereço no vetor de ponteiros.

Considerando o vetor de ponteiros declarado acima como uma variável global, podemos ilustrar a implementação de algumas funcionalidades para manipular nossa tabela de alunos. Inicialmente, vamos considerar uma função de inicialização. Uma posição do vetor estará vazia, isto é, disponível para armazenar informações de um novo aluno, se o valor do seu elemento for o ponteiro nulo. Portanto, numa função de inicialização, podemos atribuir `NULL` a todos os elementos da tabela, significando que temos, a princípio, uma tabela vazia.

```
void inicializa (void)
{
    int i;
    for (i=0; i<MAX; i++)
        tab[i] = NULL;
}
```

Uma segunda funcionalidade que podemos prever armazena os dados de um novo aluno numa posição do vetor. Vamos considerar que os dados serão fornecidos via teclado e que uma posição onde os dados serão armazenados será passada para a função. Se a posição da tabela estiver vazia, devemos alocar uma nova estrutura; caso contrário, atualizamos a estrutura já apontada pelo ponteiro.

```
void preenche (int i)
{
    if (tab[i]==NULL)
        tab[i] = (PAluno)malloc(sizeof(Aluno));

    printf("Entre com o nome:");
    scanf(" %80[^\\n]", tab[i]->nome);
    printf("Entre com a matricula:");
    scanf("%d", &tab[i]->mat);
    printf("Entre com o endereco:");
    scanf(" %120[^\\n]", tab[i]->end);
    printf("Entre com o telefone:");
    scanf(" %20[^\\n]", tab[i]->tel);
}
```

¹ Provavelmente o tipo ocupará mais espaço, pois os dados têm que estar alinhados para serem armazenados na memória.

Podemos também prever uma função para remover os dados de um aluno da tabela. Vamos considerar que a posição da tabela a ser liberada será passada para a função:

```
void remove (int i)
{
    if (tab[i] != NULL)
    {
        free(tab[i]);
        tab[i] = NULL;
    }
}
```

Para consultarmos os dados, vamos considerar uma função que imprime os dados armazenados numa determinada posição do vetor:

```
void imprime (int i)
{
    if (tab[i] != NULL)
    {
        printf("Nome: %s\n", tab[i]->nome);
        printf("Matrícula: %d\n", tab[i]->mat);
        printf("Endereço: %s\n", tab[i]->end);
        printf("Telefone: %s\n", tab[i]->tel);
    }
}
```

Por fim, podemos implementar uma função que imprima os dados de todos os alunos da tabela:

```
void imprime_tudo (void)
{
    int i;
    for (i=0; i<MAX; i++)
        imprime(i);
}
```

Exercício. Faça um programa que utilize as funções da tabela de alunos escritas acima.

Exercício. Re-escreva as funções acima sem usar uma variável global.

Sugestão: Crie um tipo Tabela e faça as funções receberem este tipo como primeiro parâmetro.

7.5. *Tipo união***

Em C, uma *união* é uma localização de memória que é compartilhada por diferentes variáveis, que podem ser de tipos diferentes. As uniões são usadas quando queremos armazenar valores heterogêneos num mesmo espaço de memória. A definição de uma união é parecida com a de uma estrutura:

```
union exemplo
{
    int i;
    char c;
}
```

Análogo à estrutura, este fragmento de código não declara nenhuma variável, apenas define o tipo união. Após uma definição, podemos declarar variáveis do tipo união:

```
union exemplo v;
```

Na variável `v`, os campos `i` e `c` compartilham o mesmo espaço de memória. A variável ocupa pelo menos o espaço necessário para armazenar o maior de seus campos (um inteiro, no caso).

O acesso aos campos de uma união é análogo ao acesso a campos de uma estrutura. Usamos o operador ponto (`.`) para acessá-los diretamente e o operador seta (`->`) para acessá-los através de um ponteiro da união. Assim, dada a declaração acima, podemos escrever:

```
v.i = 10;
```

ou

```
v.c = 'x';
```

Salientamos, no entanto, que apenas um único elemento de uma união pode estar armazenado num determinado instante, pois a atribuição a um campo da união sobrescreve o valor anteriormente atribuído a qualquer outro campo.

7.6. *Tipo enumeração***

Uma enumeração é um conjunto de constantes inteiros com nomes que especifica os valores legais que uma variável daquele tipo pode ter. É uma forma mais elegante de organizar valores constantes. Como exemplo, consideremos a criação de um tipo booleano. Variáveis deste tipo podem receber os valores 0 (FALSE) ou 1 (TRUE).

Poderíamos definir duas constantes simbólicas dissociadas e usar um inteiro para representar o tipo booleano:

```
#define FALSE 0
#define TRUE 1

typedef int Bool;
```

Desta forma, as definições de FALSE e TRUE permitem a utilização destes símbolos no código, dando maior clareza, mas o tipo booleano criado, como é equivalente a um inteiro qualquer, pode armazenar qualquer valor inteiro, não apenas FALSE e TRUE, que seria mais adequado. Para validarmos os valores atribuídos, podemos enumerar os valores constantes que um determinado tipo pode assumir, usando `enum`:

```
enum bool {
    FALSE,
    TRUE
};

typedef enum bool Bool;
```

Com isto, definimos as constantes FALSE e TRUE. Por *default*, o primeiro símbolo representa o valor 0, o seguinte o valor 1, e assim por diante. Poderíamos explicitar os valores dos símbolos numa enumeração, como por exemplo:

```
enum bool {  
    TRUE = 1,  
    FALSE = 0,  
};
```

No exemplo do tipo booleano, a numeração *default* coincide com a desejada (desde que o símbolo FALSE preceda o símbolo TRUE dentro da lista da enumeração).

A declaração de uma variável do tipo criado pode ser dada por:

```
Bool resultado;
```

onde resultado representa uma variável que pode receber apenas os valores FALSE (0) ou TRUE (1).

8. Matrizes

W. Celes e J. L. Rangel

Já discutimos em capítulos anteriores a construção de conjuntos unidimensionais através do uso de vetores. A linguagem C também permite a construção de conjuntos bi ou multidimensionais. Neste capítulo, discutiremos em detalhe a manipulação de matrizes, representadas por conjuntos bidimensionais de valores numéricos. As construções apresentadas aqui podem ser estendidas para conjuntos de dimensões maiores.

8.1. Alocação estática versus dinâmica

Antes de tratarmos das construções de matrizes, vamos recapitular alguns conceitos apresentados com vetores. A forma mais simples de declararmos um vetor de inteiros em C é mostrada a seguir:

```
int v[10];
```

ou, se quisermos criar uma constante simbólica para a dimensão:

```
#define N 10  
int v[N];
```

Podemos dizer que, nestes casos, os vetores são declarados “estaticamente”¹. A variável que representa o vetor é uma *constante* que armazena o endereço ocupado pelo primeiro elemento do vetor. Esses vetores podem ser declarados como variáveis globais ou dentro do corpo de uma função. Se declarado dentro do corpo de uma função, o vetor existirá apenas enquanto a função estiver sendo executada, pois o espaço de memória para o vetor é reservado na pilha de execução. Portanto, não podemos fazer referência ao espaço de memória de um vetor local de uma função que já retornou.

O problema de declararmos um vetor estaticamente, seja como variável global ou local, é que precisamos saber de antemão a dimensão máxima do vetor. Usando alocação dinâmica, podemos determinar a dimensão do vetor em tempo de execução:

```
int* v;  
...  
v = (int*) malloc(n * sizeof(int));
```

Neste fragmento de código, n representa uma variável com a dimensão do vetor, determinada em tempo de execução (podemos, por exemplo, capturar o valor de n fornecido pelo usuário). Após a alocação dinâmica, acessamos os elementos do vetor da mesma forma que os elementos de vetores criados estaticamente. Outra diferença importante: com alocação dinâmica, declaramos uma variável do tipo ponteiro que posteriormente recebe o valor do endereço do primeiro elemento do vetor, alocado dinamicamente. A área de memória ocupada pelo vetor permanece válida até que seja explicitamente liberada (através da função free). Portanto, mesmo que um vetor seja

¹ O termo “estático” aqui refere-se ao fato de não usarmos alocação dinâmica.

criado dinamicamente dentro da função, podemos acessá-lo depois da função ser finalizada, pois a área de memória ocupada por ele permanece válida, isto é, o vetor não está alocado na pilha de execução. Usamos esta propriedade quando escrevemos a função que duplica uma cadeia de caracteres (*string*): a função `duplica` aloca um vetor de `char` dinamicamente, preenche seus valores e retorna o ponteiro, para que a função que chama possa acessar a nova cadeia de caracteres.

A linguagem C oferece ainda um mecanismo para re-alocarmos um vetor dinamicamente. Em tempo de execução, podemos verificar que a dimensão inicialmente escolhida para um vetor tornou-se insuficiente (ou excessivamente grande), necessitando um redimensionamento. A função `realloc` da biblioteca padrão nos permite re-alocar um vetor, preservando o conteúdo dos elementos, que permanecem válidos após a re-alocação (no fragmento de código abaixo, `m` representa a nova dimensão do vetor).

```
v = (int*) realloc(v, m*sizeof(int));
```

Vale salientar que, sempre que possível, optamos por trabalhar com vetores criados estaticamente. Eles tendem a ser mais eficientes, já que os vetores alocados dinamicamente têm uma indireção a mais (primeiro acessa-se o valor do endereço armazenado na variável ponteiro para então acessar o elemento do vetor).

8.2. Vetores bidimensionais – Matrizes

A linguagem C permite a criação de vetores bidimensionais, declarados estaticamente. Por exemplo, para declararmos uma matriz de valores reais com 4 linhas e 3 colunas, fazemos:

```
float mat[4][3];
```

Esta declaração reserva um espaço de memória necessário para armazenar os 12 elementos da matriz, que são armazenados de maneira contínua, organizados linha a linha.

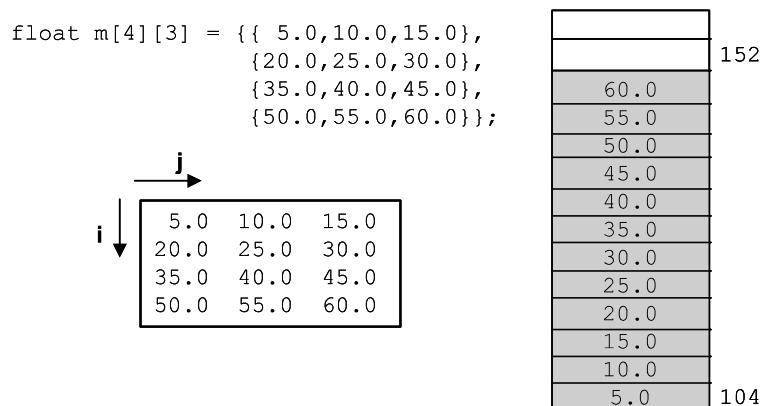


Figura 8.1: Alocação dos elementos de uma matriz.

Os elementos da matriz são acessados com indexação dupla: `mat[i][j]`. O primeiro índice, `i`, acessa a linha e o segundo, `j`, acessa a coluna. Como em C a indexação começa em zero, o elemento da primeira linha e primeira coluna é acessado por `mat[0][0]`. Após a declaração estática de uma matriz, a variável que representa a matriz, `mat` no exemplo acima, representa um ponteiro para o primeiro “vetor-linha”, composto por 3 elementos. Com isto, `mat[1]` aponta para o primeiro elemento do segundo “vetor-linha”, e assim por diante.

As matrizes também podem ser inicializadas na declaração:

```
float mat[4][3] = {{1,2,3},{4,5,6},{7,8,9},{10,11,12}};
```

Ou podemos inicializar seqüencialmente:

```
float mat[4][3] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

O número de elementos por linha pode ser omitido numa inicialização, mas o número de colunas deve, obrigatoriamente, ser fornecido:

```
float mat[][3] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

Passagem de matrizes para funções

Conforme dissemos acima, uma matriz criada estaticamente é representada por um ponteiro para um “vetor-linha” com o número de elementos da linha. Quando passamos uma matriz para uma função, o parâmetro da função deve ser deste tipo. Infelizmente, a sintaxe para representar este tipo é obscura. O protótipo de uma função que recebe a matriz declarada acima seria:

```
void f (... , float (*mat)[3], ...);
```

Uma segunda opção é declarar o parâmetro como matriz, podendo omitir o número de linhas²:

```
void f (... , float mat[][3], ...);
```

De qualquer forma, o acesso aos elementos da matriz dentro da função é feito da forma usual, com indexação dupla.

Na próxima seção, examinaremos formas de trabalhar com matrizes alocadas dinamicamente. No entanto, vale salientar que recomendamos, sempre que possível, o uso de matrizes alocadas estaticamente. Em diversas aplicações, as matrizes têm dimensões fixas e não justificam a criação de estratégias para trabalhar com alocação dinâmica. Em aplicações da área de Computação Gráfica, por exemplo, é comum trabalharmos com matrizes de 4 por 4 para representar transformações geométricas e projeções. Nestes casos, é muito mais simples definirmos as matrizes estaticamente (`float mat[4][4];`), uma

² Isto também vale para vetores. Um protótipo de uma função que recebe um vetor como parâmetro pode ser dado por: `void f (... , float v[], ...);`.

vez que sabemos de antemão as dimensões a serem usadas. Nestes casos, vale a pena definirmos um tipo próprio, pois nos livraremos das construções sintáticas confusas explicitadas acima. Por exemplo, podemos definir o tipo `Matrix4`.

```
typedef float Matrix4[4][4];
```

Com esta definição podemos declarar variáveis e parâmetros deste tipo:

```
Matrix4 m;                                /* declaração de variável */
...
void f (... , Matrix4 m, ...);    /* especificação de parâmetro */
```

8.3. Matrizes dinâmicas

As matrizes declaradas estaticamente sofrem das mesmas limitações dos vetores: precisamos saber de antemão suas dimensões. O problema que encontramos é que a linguagem C só permite alocarmos dinamicamente conjuntos unidimensionais. Para trabalharmos com matrizes alocadas dinamicamente, temos que criar abstrações conceituais com vetores para representar conjuntos bidimensionais. Nesta seção, discutiremos duas estratégias distintas para representar matrizes alocadas dinamicamente.

Matriz representada por um vetor simples

Conceitualmente, podemos representar uma matriz num vetor simples. Reservamos as primeiras posições do vetor para armazenar os elementos da primeira linha, seguidos dos elementos da segunda linha, e assim por diante. Como, de fato, trabalharemos com um vetor unidimensional, temos que criar uma disciplina para acessar os elementos da matriz, representada conceitualmente. A estratégia de endereçamento para acessar os elementos é a seguinte: se quisermos acessar o que seria o elemento `mat[i][j]` de uma matriz, devemos acessar o elemento `v[i*n+j]`, onde `n` representa o número de colunas da matriz.

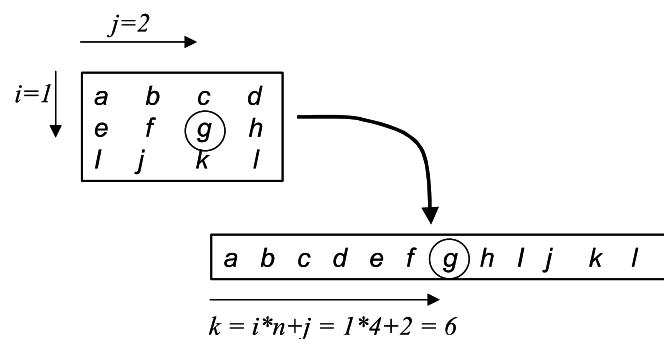


Figura 8.2: Matriz representada por vetor simples.

Esta conta de endereçamento é intuitiva: se quisermos acessar elementos da terceira ($i=2$) linha da matriz, temos que pular duas linhas de elementos ($i*n$) e depois indexar o elemento da linha com j .

Com esta estratégia, a alocação da “matriz” recai numa alocação de vetor que tem $m \times n$ elementos, onde m e n representam as dimensões da matriz.

```
float *mat;           /* matriz representada por um vetor */
...
mat = (float*) malloc(m*n*sizeof(float));
...
```

No entanto, somos obrigados a usar uma notação desconfortável, $v[i \cdot n + j]$, para acessar os elementos, o que pode deixar o código pouco legível.

Matriz representada por um vetor de ponteiros

Nesta segunda estratégia, faremos algo parecido com o que fizemos para tratar vetores de cadeias de caracteres, que em C são representados por conjuntos bidimensionais de caracteres. De acordo com esta estratégia, cada linha da matriz é representada por um vetor independente. A matriz é então representada por um vetor de vetores, ou vetor de ponteiros, no qual cada elemento armazena o endereço do primeiro elemento de cada linha. A figura abaixo ilustra o arranjo da memória utilizada nesta estratégia.

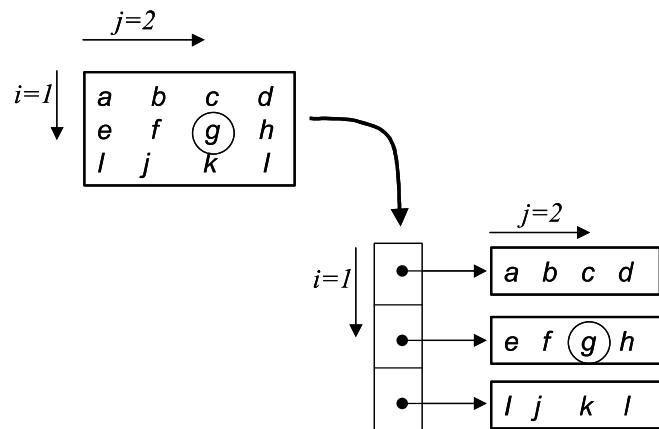


Figura 8.3: Matriz com vetor de ponteiros.

A alocação da matriz agora é mais elaborada. Primeiro, temos que alocar o vetor de ponteiros. Em seguida, alocamos cada uma das linhas da matriz, atribuindo seus endereços aos elementos do vetor de ponteiros criado. O fragmento de código abaixo ilustra esta codificação:

```
int i;
float **mat;           /* matriz representada por um vetor de ponteiros */
...
mat = (float**) malloc(m*sizeof(float*));
for (i=0; i<m; i++)
    m[i] = (float*) malloc(n*sizeof(float));
```

A grande vantagem desta estratégia é que o acesso aos elementos é feito da mesma forma que quando temos uma matriz criada estaticamente, pois, se `mat` representa uma matriz

alocada segundo esta estratégia, `mat[i]` representa o ponteiro para o primeiro elemento da linha `i`, e, consequentemente, `mat[i][j]` acessa o elemento da coluna `j` da linha `i`.

A liberação do espaço de memória ocupado pela matriz também exige a construção de um laço, pois temos que liberar cada linha antes de liberar o vetor de ponteiros:

```
...
for (i=0; i<m; i++)
    free(mat[i]);
free(mat);
```

8.4. Representação de matrizes

Para exemplificar o uso de matrizes dinâmicas, vamos discutir a escolha de um tipo para representar as matrizes e um conjunto de operações implementadas sobre o tipo escolhido. Podemos considerar, por exemplo, a implementação de funções básicas, sobre as quais podemos futuramente implementar funções mais complexas, tais como soma, multiplicação e inversão de matrizes.

Vamos considerar a implementação das seguintes operações básicas:

- `cria`: operação que cria uma matriz de dimensão `m` por `n`;
- `libera`: operação que libera a memória alocada para a matriz;
- `acessa`: operação que acessa o elemento da linha `i` e da coluna `j` da matriz;
- `atribui`: operação que atribui o elemento da linha `i` e da coluna `j` da matriz.

A seguir, mostraremos a implementação dessas operações usando as duas estratégias para alocar dinamicamente uma matriz, apresentadas na seção anterior.

Matriz com vetor simples

Usando a estratégia com um vetor simples, o tipo matriz pode ser representado por uma estrutura que guarda a dimensão da matriz e o vetor que armazena os elementos.

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

typedef struct matriz Matriz;
```

A função que cria a matriz dinamicamente deve alocar a estrutura que representa a matriz e alocar o vetor dos elementos:

```
Matriz* cria (int m, int n)
{
    Matriz* mat = (Matriz*) malloc(sizeof(Matriz));
    mat->lin = m;
    mat->col = n;
    mat->v = (float*) malloc(m*n*sizeof(float));
```

```

        return mat;
    }
}

```

Poderíamos ainda incluir na criação uma inicialização dos elementos da matriz, por exemplo atribuindo-lhes valores iguais a zero.

A função que libera a memória deve liberar o vetor de elementos e então liberar a estrutura que representa a matriz:

```

void libera (Matriz* mat)
{
    free(mat->v);
    free(mat);
}

```

A função de acesso e atribuição pode fazer um teste adicional para garantir que não haja invasão de memória. Se a aplicação que usa o módulo tentar acessar um elemento fora das dimensões da matriz, podemos reportar um erro e abortar o programa. A implementação destas funções pode ser dada por:

```

float acessa (Matriz* mat, int i, int j)
{
    int k; /* índice do elemento no vetor */

    if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Acesso inválido!\n");
        exit(1);
    }
    k = i*mat->col + j;
    return mat->v[k];
}

void atribui (Matriz* mat, int i, int j, float v)
{
    int k; /* índice do elemento no vetor */

    if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Atribuição inválida!\n");
        exit(1);
    }
    k = i*mat->col + j;
    mat->v[k] = v;
}

```

Matriz com vetor de ponteiros

O módulo de implementação usando a estratégia de representar a matriz por um vetor de ponteiros é apresentado a seguir. O tipo que representa a matriz, neste caso, pode ser dado por:

```

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

```

```
typedef struct matriz Matriz;
```

As funções para criar uma nova matriz e para liberar uma matriz previamente criada podem ser dadas por:

```
Matriz* cria (int m, int n)
{
    int i;
    Matriz mat = (Matriz*) malloc(sizeof(Matriz));
    mat->lin = m;
    mat->col = n;
    mat->v = (float**) malloc(m*sizeof(float*));
    for (i=0; i<m; i++)
        mat->v[i] = (float*) malloc(n*sizeof(float));
    return mat;
}

void libera (Matriz* mat)
{
    int i;
    for (i=0; i<mat->lin; i++)
        free(mat->v[i]);
    free(mat->v);
    free(mat);
}
```

As funções para acessar e atribuir podem ser implementadas conforme ilustrado abaixo:

```
float acessa (Matriz* mat, int i, int j)
{
    if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Acesso inválido!\n");
        exit(1);
    }
    return mat->v[i][j];
}

void atribui (Matriz* mat, int i, int j, float v)
{
    if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Atribuição inválida!\n");
        exit(1);
    }
    mat->v[i][j] = v;
}
```

Exercício: Escreva um programa que faça uso das operações de matriz definidas acima. Note que a estratégia de implementação não deve alterar o uso das operações.

Exercício: Implemente uma função que, dada uma matriz, crie dinamicamente a matriz transposta correspondente, fazendo uso das operações básicas discutidas acima.

Exercício: Implemente uma função que determine se uma matriz é ou não simétrica quadrada, também fazendo uso das operações básicas.

8.5. Representação de matrizes simétricas

Em uma matriz simétrica n por n , não há necessidade, no caso de $i \neq j$, de armazenar ambos os elementos $\text{mat}[i][j]$ e $\text{mat}[j][i]$, porque os dois têm o mesmo valor. Portanto, basta guardar os valores dos elementos da diagonal e de metade dos elementos restantes – por exemplo, os elementos abaixo da diagonal, para os quais $i > j$. Ou seja, podemos fazer uma economia de espaço usado para alocar a matriz. Em vez de n^2 valores, podemos armazenar apenas s elementos, sendo s dado por:

$$s = n + \frac{(n^2 - n)}{2} = \frac{n(n+1)}{2}$$

Podemos também determinar s como sendo a soma de uma progressão aritmética, pois temos que armazenar um elemento da primeira linha, dois elementos da segunda, três da terceira, e assim por diante.

$$s = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

A implementação deste tipo abstrato também pode ser feita com um vetor simples ou um vetor de ponteiros. A seguir, discutimos a implementação das operações para criar uma matriz e para acessar os elementos, agora para um tipo que representa uma matriz simétrica.

Matriz simétrica com vetor simples

Usando um vetor simples para armazenar os elementos da matriz, dimensionamos o vetor com apenas s elementos. A estrutura que representa a matriz pode ser dada por:

```
struct matsim {
    int dim;           /* matriz obrigatoriamente quadrada */
    float* v;
};

typedef struct matsim MatSim;
```

Uma função para criar uma matriz simétrica pode ser dada por:

```
MatSim* cria (int n)
{
    int s = n*(n+1)/2;
    MatSim* mat = (MatSim*) malloc(sizeof(MatSim));
    mat->dim = n;
    mat->v = (float*) malloc(s*sizeof(float));
    return mat;
}
```

O acesso aos elementos da matriz deve ser feito como se estivéssemos representando a matriz inteira. Se for um acesso a um elemento acima da diagonal ($i < j$), o valor de retorno é o elemento simétrico da parte inferior, que está devidamente representado. O endereçamento de um elemento da parte inferior da matriz é feito saltando-se os elementos das linhas superiores. Assim, se desejarmos acessar um elemento da quinta linha ($i=4$),

devemos saltar $1+2+3+4$ elementos, isto é, devemos saltar $1+2+\dots+i$ elementos, ou seja, $i*(i+1)/2$ elementos. Depois, usamos o índice j para acessar a coluna.

```
float acessa (MatSim* mat, int i, int j)
{
    int k; /* indice do elemento no vetor */

    if (i<0 || i>=mat->dim || j<0 || j>=mat->dim) {
        printf("Acesso inválido!\n");
        exit(1);
    }
    if (i>=j)
        k = i*(i+1)/2 + j;
    else
        k = j*(j+1)/2 + i;
    return mat->v[k];
}
```

Matriz simétrica com vetor de ponteiros

A estratégia de trabalhar com vetores de ponteiros para matrizes alocadas dinamicamente é muito adequada para a representação matrizes simétricas. Numa matriz simétrica, para otimizar o uso da memória, armazenamos apenas a parte triangular inferior da matriz. Isto significa que a primeira linha será representada por um vetor de um único elemento, a segunda linha será representada por um vetor de dois elementos e assim por diante. Como o uso de um vetor de ponteiros trata as linhas como vetores independentes, a adaptação desta estratégia para matrizes simétricas fica simples.

O tipo da matriz pode ser definido por:

```
struct matsim {
    int dim;
    float** v;
};

typedef struct matsim MatSim;
```

Para criar a matriz, basta alocarmos um número variável de elementos para cada linha. O código abaixo ilustra uma possível implementação:

```
MatSim* cria (int n)
{
    int i;
    MatSim* mat = (MatSim*) malloc(sizeof(MatSim));
    mat->dim = n;
    mat->v = (float**) malloc(n*sizeof(float*));
    for (i=0; i<n; i++)
        mat->v[i] = (float*) malloc((i+1)*sizeof(float));
    return mat;
}
```

O acesso aos elementos é natural, desde que tenhamos o cuidado de não acessar elementos que não estejam explicitamente alocados (isto é, elementos com $i < j$).

```

float acessa (MatSim* mat, int i, int j)
{
    if (i<0 || i>=mat->dim || j<0 || j>=mat->dim) {
        printf("Acesso inválido!\n");
        exit(1);
    }
    if (i>=j)
        return mat->v[i][j];
    else
        return mat->v[j][i];
}

```

Finalmente, observamos que exatamente as mesmas técnicas poderiam ser usadas para representar uma matriz “triangular”, isto é, uma matriz cujos elementos acima (ou abaixo) da diagonal são todos nulos. Neste caso, a principal diferença seria na função `acessa`, que teria como resultado o valor zero em um dos lados da diagonal, em vez acessar o valor simétrico.

Exercício: Escreva um código para representar uma matriz triangular inferior.

Exercício: Escreva um código para representar uma matriz triangular superior.

9. Tipos Abstratos de Dados

R. Cerqueira, W. Celes e J.L. Rangel

Neste capítulo, discutiremos uma importante técnica de programação baseada na definição de *Tipos Abstratos de Dados* (TAD). Veremos também como a linguagem C pode nos ajudar na implementação de um TAD, através de alguns de seus mecanismos básicos de *modularização* (divisão de um programa em vários arquivos fontes).

9.1. Módulos e Compilação em Separado

Como foi visto no capítulo 1, um programa em C pode ser dividido em vários arquivos fontes (arquivos com extensão .c). Quando temos um arquivo com funções que representam apenas parte da implementação de um programa completo, denominamos esse arquivo de *módulo*. Assim, a implementação de um programa pode ser composta por um ou mais módulos.

No caso de um programa composto por vários módulos, cada um desses módulos deve ser compilado separadamente, gerando um arquivo objeto (geralmente um arquivo com extensão .o ou .obj) para cada módulo. Após a compilação de todos os módulos, uma outra ferramenta, denominada *ligador*, é usada para juntar todos os arquivos objeto em um único arquivo executável.

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

Para ilustrar o uso de módulos em C, considere que temos um arquivo str.c que contém apenas a implementação das funções de manipulação de strings comprimento, copia e concatena vistas no capítulo 6. Considere também que temos um arquivo progl.c com o seguinte código:

```
#include <stdio.h>

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

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

A partir desses dois arquivos fontes, podemos gerar um programa executável compilando cada um dos arquivos separadamente e depois ligando-os em um único

arquivo executável. Por exemplo, com o compilador Gnu C (gcc) utilizariamos a seguinte seqüência de comandos para gerar o arquivo executável `prog1.exe`:

```
> gcc -c str.c
> gcc -c prog1.c
> gcc -o prog1.exe str.o prog1.o
```

O mesmo arquivo `str.c` pode ser usado para compor outros programas que queiram utilizar suas funções. Para que as funções implementadas em `str.c` possam ser usadas por um outro módulo C, este precisa conhecer os cabeçalhos das funções oferecidas por `str.c`. No exemplo anterior, isso foi resolvido pela repetição dos cabeçalhos das funções no início do arquivo `prog1.c`. Entretanto, para módulos que ofereçam várias funções ou que queiram usar funções de muitos outros módulos, essa repetição manual pode ficar muito trabalhosa e sensível a erros. Para contornar esse problema, todo módulo de funções C costuma ter associado a ele um arquivo que contém apenas os cabeçalhos das funções oferecidas pelo módulo e, eventualmente, os tipos de dados que ele exporta (`typedef`'s, `struct`'s, etc). Esse arquivo de cabeçalhos segue o mesmo nome do módulo ao qual está associado, só que com a extensão `.h`. Assim, poderíamos definir um arquivo `str.h` para o módulo do exemplo anterior, com o seguinte conteúdo:

```
/* Funções oferecidas pelo modulo str.c */

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

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

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

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

Agora, ao invés de repetir manualmente os cabeçalhos dessas funções, todo módulo que quiser usar as funções de `str.c` precisa apenas incluir o arquivo `str.h`. No exemplo anterior, o módulo `prog1.c` poderia ser simplificado da seguinte forma:

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

int main (void) {
    char str[101], str1[51], str2[51];
    printf("Entre com uma seqüência de caracteres: ");
    scanf(" %50[^\\n]", str1);
    printf("Entre com outra seqüência de caracteres: ");
    scanf(" %50[^\\n]", str2);
```

```

copia(str, str1);
concatena(str, str2);
printf("Comprimento da concatenação: %d\n", comprimento(str));
return 0;
}

```

Note que os arquivos de cabeçalhos das funções da biblioteca padrão do C (que acompanham seu compilador) são incluídos da forma `#include <arquivo.h>`, enquanto que os arquivos de cabeçalhos dos seus módulos são geralmente incluídos da forma `#include "arquivo.h"`. O uso dos delimitadores `< >` e `" "` indica para o compilador onde ele deve procurar esses arquivos de cabeçalhos durante a compilação.

9.2. *Tipo Abstrato de Dados*

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

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

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

Exemplo 1: TAD Ponto

Como nosso primeiro exemplo de TAD, vamos considerar a criação de um tipo de dado para representar um ponto no \mathbb{R}^2 . Para isso, devemos definir um tipo abstrato, que denominaremos de `Ponto`, e o conjunto de funções que operam sobre esse tipo. Neste exemplo, vamos considerar as seguintes operações:

- `cria`: operação que cria um ponto com coordenadas `x` e `y`;
- `libera`: operação que libera a memória alocada por um ponto;
- `acessa`: operação que devolve as coordenadas de um ponto;
- `atribui`: operação que atribui novos valores às coordenadas de um ponto;
- `distancia`: operação que calcula a distância entre dois pontos.

A interface desse módulo pode ser dada pelo código a seguir:

Arquivo ponto.h:

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

/* Tipo exportado */
typedef struct ponto Ponto;

/* Funções exportadas */

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

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

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

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

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

Note que a composição da estrutura `Ponto` (`struct ponto`) não é exportada pelo módulo. Dessa forma, os demais módulos que usarem esse TAD não poderão acessar diretamente os campos dessa estrutura. Os *clientes* desse TAD só terão acesso às informações que possam ser obtidas através das funções exportadas pelo arquivo `ponto.h`.

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

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

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

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

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

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

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

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

As funções para acessar e atribuir valores às coordenadas de um ponto são de fácil implementação, como pode ser visto a seguir.

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

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

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

Exercício: Escreva um programa que faça uso do TAD ponto definido acima.

Exercício: Acrescente novas operações ao TAD ponto, tais como soma e subtração de pontos.

Exercício: Acrescente novas operações ao TAD ponto, de tal forma que seja possível obter uma representação do ponto em coordenadas polares.

Exercício: Implemente um novo TAD para representar pontos no \mathbb{R}^3 .

Exemplo 2: TAD Matriz

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

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

A interface do módulo pode ser dada pelo código abaixo:

Arquivo matriz.h:

```
/* TAD: matriz m por n */

/* Tipo exportado */
typedef struct matriz Matriz;

/* Funções exportadas */

/* Função cria
** Aloca e retorna uma matriz de dimensão m por n
*/
Matriz* cria (int m, int n);

/* Função libera
** Libera a memória de uma matriz previamente criada.
*/
void libera (Matriz* mat);

/* Função acessa
** Retorna o valor do elemento da linha i e coluna j da matriz
*/
float acessa (Matriz* mat, int i, int j);

/* Função atribui
** Atribui o valor dado ao elemento da linha i e coluna j da matriz
*/
void atribui (Matriz* mat, int i, int j, float v);

/* Função linhas
** Retorna o número de linhas da matriz
*/
int linhas (Matriz* mat);
```

```

/* Função colunas
** Retorna o número de colunas da matriz
*/
int colunas (Matriz* mat);

```

A seguir, mostraremos a implementação deste tipo abstrato usando as duas estratégias apresentadas no capítulo 8: matrizes dinâmicas representadas por vetores simples e matrizes dinâmicas representadas por vetores de ponteiros. A interface do módulo independe da estratégia de implementação adotada, o que é altamente desejável, pois podemos mudar a implementação sem afetar as aplicações que fazem uso do tipo abstrato. O arquivo `matriz1.c` apresenta a implementação através de vetor simples e o arquivo `matriz2.c` apresenta a implementação através de vetor de ponteiros.

Arquivo matriz1.c:

```

#include <stdlib.h>          /* malloc, free, exit */
#include <stdio.h>           /* printf */
#include "matriz.h"

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

Matriz* cria (int m, int n) {
    Matriz* mat = (Matriz*) malloc(sizeof(Matriz));
    if (mat == NULL) {
        printf("Memória insuficiente!\n");
        exit(1);
    }
    mat->lin = m;
    mat->col = n;
    mat->v = (float*) malloc(m*n*sizeof(float));
    return mat;
}

void libera (Matriz* mat){
    free(mat->v);
    free(mat);
}

float acessa (Matriz* mat, int i, int j) {
    int k;      /* índice do elemento no vetor */

    if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Acesso inválido!\n");
        exit(1);
    }
    k = i*mat->col + j;
    return mat->v[k];
}

void atribui (Matriz* mat, int i, int j, float v) {
    int k;      /* índice do elemento no vetor */

    if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Atribuição inválida!\n");
        exit(1);
    }
}

```

```

    k = i*mat->col + j;
    mat->v[k] = v;
}

int linhas (Matriz* mat) {
    return mat->lin;
}

int colunas (Matriz* mat) {
    return mat->col;
}

```

Arquivo matriz2.c:

```

#include <stdlib.h>      /* malloc, free, exit */
#include <stdio.h>        /* printf */
#include "matriz.h"

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

Matriz* cria (int m, int n) {
    int i;
    Matriz* mat = (Matriz*) malloc(sizeof(Matriz));
    if (mat == NULL) {
        printf("Memória insuficiente!\n");
        exit(1);
    }
    mat->lin = m;
    mat->col = n;
    mat->v = (float**) malloc(m*sizeof(float*));
    for (i=0; i<m; i++)
        mat->v[i] = (float*) malloc(n*sizeof(float));
    return mat;
}

void libera (Matriz* mat) {
    int i;
    for (i=0; i<mat->lin; i++)
        free(mat->v[i]);
    free(mat->v);
    free(mat);
}

float acessa (Matriz* mat, int i, int j) {
    if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Acesso inválido!\n");
        exit(1);
    }
    return mat->v[i][j];
}

void atribui (Matriz* mat, int i, int j, float v) {
    if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Atribuição inválida!\n");
        exit(1);
    }
    mat->v[i][j] = v;
}

int linhas (Matriz* mat) {
    return mat->lin;
}

```

```
int colunas (Matriz* mat) {  
    return mat->col;  
}
```

Exercício: Escreva um programa que faça uso do TAD matriz definido acima. Teste o seu programa com as duas implementações vistas.

Exercício: Usando apenas as operações definidas pelo TAD matriz, implemente uma função que determine se uma matriz é ou não quadrada simétrica.

Exercício: Usando apenas as operações definidas pelo TAD matriz, implemente uma função que, dada uma matriz, crie dinamicamente a matriz transposta correspondente.

Exercício: Defina um TAD para implementar matrizes quadradas simétricas, de acordo com a representação sugerida no capítulo 8.

10. Listas Encadeadas

W. Celes e J. L. Rangel

Para representarmos um grupo de dados, já vimos que podemos usar um vetor em C. O vetor é a forma mais primitiva de representar diversos elementos agrupados. Para simplificar a discussão dos conceitos que serão apresentados agora, vamos supor que temos que desenvolver uma aplicação que deve representar um grupo de valores inteiros. Para tanto, podemos declarar um vetor escolhendo um número máximo de elementos.

```
#define MAX 1000  
int vet[MAX];
```

Ao declararmos um vetor, reservamos um espaço contíguo de memória para armazenar seus elementos, conforme ilustra a figura abaixo.

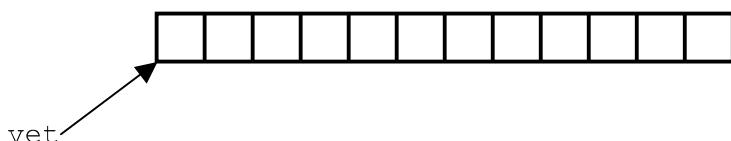


Figura 9.1: Um vetor ocupa um espaço contíguo de memória, permitindo que qualquer elemento seja acessado indexando-se o ponteiro para o primeiro elemento.

O fato de o vetor ocupar um espaço contíguo na memória nos permite acessar qualquer um de seus elementos a partir do ponteiro para o primeiro elemento. De fato, o símbolo `vet`, após a declaração acima, como já vimos, representa um ponteiro para o primeiro elemento do vetor, isto é, o valor de `vet` é o endereço da memória onde o primeiro elemento do vetor está armazenado. De posse do ponteiro para o primeiro elemento, podemos acessar qualquer elemento do vetor através do operador de indexação `vet[i]`. Dizemos que o vetor é uma estrutura que possibilita acesso randômico aos elementos, pois podemos acessar qualquer elemento aleatoriamente.

No entanto, o vetor não é uma estrutura de dados muito flexível, pois precisamos dimensioná-lo com um número máximo de elementos. Se o número de elementos que precisarmos armazenar exceder a dimensão do vetor, teremos um problema, pois não existe uma maneira simples e barata (computacionalmente) para alterarmos a dimensão do vetor em tempo de execução. Por outro lado, se o número de elementos que precisarmos armazenar no vetor for muito inferior à sua dimensão, estaremos subutilizando o espaço de memória reservado.

A solução para esses problemas é utilizar estruturas de dados que cresçam à medida que precisarmos armazenar novos elementos (e diminuam à medida que precisarmos retirar elementos armazenados anteriormente). Tais estruturas são chamadas *dinâmicas* e armazenam cada um dos seus elementos usando alocação dinâmica.

Nas seções a seguir, discutiremos a estrutura de dados conhecida como *lista encadeada*. As listas encadeadas são amplamente usadas para implementar diversas outras estruturas de dados com semânticas próprias, que serão tratadas nos capítulos seguintes.

10.1. Lista encadeada

Numa lista encadeada, para cada novo elemento inserido na estrutura, alocamos um espaço de memória para armazená-lo. Desta forma, o espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenado. No entanto, não podemos garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo, portanto não temos acesso direto aos elementos da lista. Para que seja possível percorrer todos os elementos da lista, devemos explicitamente guardar o encadeamento dos elementos, o que é feito armazenando-se, junto com a informação de cada elemento, um ponteiro para o próximo elemento da lista. A Figura 9.2 ilustra o arranjo da memória de uma lista encadeada.

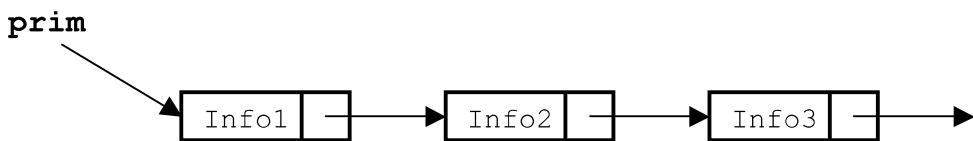


Figura 9.2: Arranjo da memória de uma lista encadeada.

A estrutura consiste numa seqüência encadeada de elementos, em geral chamados de *nós da lista*. A lista é representada por um ponteiro para o primeiro elemento (ou nó). Do primeiro elemento, podemos alcançar o segundo seguindo o encadeamento, e assim por diante. O último elemento da lista aponta para `NULL`, sinalizando que não existe um próximo elemento.

Para exemplificar a implementação de listas encadeadas em C, vamos considerar um exemplo simples em que queremos armazenar valores inteiros numa lista encadeada. O nó da lista pode ser representado pela estrutura abaixo:

```
struct lista {  
    int info;  
    struct lista* prox;  
};  
  
typedef struct lista Lista;
```

Devemos notar que trata-se de uma estrutura auto-referenciada, pois, além do campo que armazena a informação (no caso, um número inteiro), há um campo que é um ponteiro para uma próxima estrutura do mesmo tipo. Embora não seja essencial, é uma boa estratégia definirmos o tipo `Lista` como sinônimo de `struct lista`, conforme ilustrado acima. O tipo `Lista` representa um nó da lista e a estrutura de lista encadeada é representada pelo ponteiro para seu primeiro elemento (tipo `Lista*`).

Considerando a definição de `Lista`, podemos definir as principais funções necessárias para implementarmos uma lista encadeada.

Função de inicialização

A função que inicializa uma lista deve criar uma lista vazia, sem nenhum elemento. Como a lista é representada pelo ponteiro para o primeiro elemento, uma lista vazia é

representada pelo ponteiro `NULL`, pois não existem elementos na lista. A função tem como valor de retorno a lista vazia inicializada, isto é, o valor de retorno é `NULL`. Uma possível implementação da função de inicialização é mostrada a seguir:

```
/* função de inicialização: retorna uma lista vazia */
Lista* inicializa (void)
{
    return NULL;
}
```

Função de inserção

Uma vez criada a lista vazia, podemos inserir novos elementos nela. Para cada elemento inserido na lista, devemos alocar dinamicamente a memória necessária para armazenar o elemento e encadeá-lo na lista existente. A função de inserção mais simples insere o novo elemento no início da lista.

Uma possível implementação dessa função é mostrada a seguir. Devemos notar que o ponteiro que representa a lista deve ter seu valor atualizado, pois a lista deve passar a ser representada pelo ponteiro para o novo primeiro elemento. Por esta razão, a função de inserção recebe como parâmetros de entrada a lista onde será inserido o novo elemento e a informação do novo elemento, e tem como valor de retorno a nova lista, representada pelo ponteiro para o novo elemento.

```
/* inserção no inicio: retorna a lista atualizada */
Lista* insere (Lista* l, int i)
{
    Lista* novo = (Lista*) malloc(sizeof(Lista));
    novo->info = i;
    novo->prox = l;
    return novo;
}
```

Esta função aloca dinamicamente o espaço para armazenar o novo nó da lista, guarda a informação no novo nó e faz este nó apontar para (isto é, ter como próximo elemento) o elemento que era o primeiro da lista. A função então retorna o novo valor que representa a lista, que é o ponteiro para o novo primeiro elemento. A Figura 9.3 ilustra a operação de inserção de um novo elemento no início da lista.

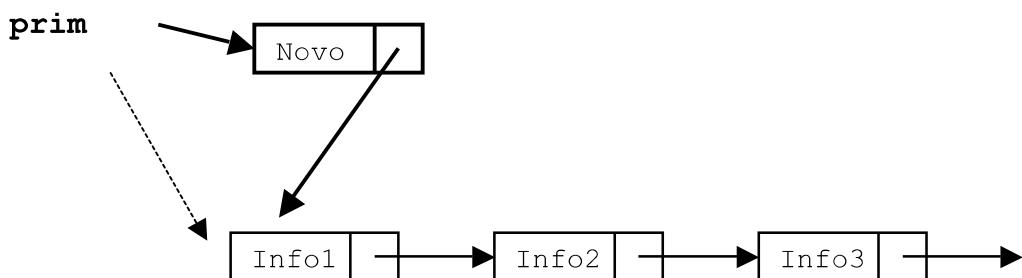


Figura 9.3: Inserção de um novo elemento no início da lista.

A seguir, ilustramos um trecho de código que cria uma lista inicialmente vazia e insere nela novos elementos.

```

int main (void)
{
    Lista* l;           /* declara uma lista não inicializada */
    l = inicializa();   /* inicializa lista como vazia */
    l = insere(l, 23); /* insere na lista o elemento 23 */
    l = insere(l, 45); /* insere na lista o elemento 45 */
    ...
    return 0;
}

```

Observe que não podemos deixar de atualizar a variável que representa a lista a cada inserção de um novo elemento.

Função que percorre os elementos da lista

Para ilustrar a implementação de uma função que percorre todos os elementos da lista, vamos considerar a criação de uma função que imprima os valores dos elementos armazenados numa lista. Uma possível implementação dessa função é mostrada a seguir.

```

/* função imprime: imprime valores dos elementos */
void imprime (Lista* l)
{
    Lista* p; /* variável auxiliar para percorrer a lista */
    for (p = l; p != NULL; p = p->prox)
        printf("info = %d\n", p->info);
}

```

Função que verifica se lista está vazia

Pode ser útil implementarmos uma função que verifique se uma lista está vazia ou não. A função recebe a lista e retorna 1 se estiver vazia ou 0 se não estiver vazia. Como sabemos, uma lista está vazia se seu valor é NULL. Uma implementação dessa função é mostrada a seguir:

```

/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int vazia (Lista* l)
{
    if (l == NULL)
        return 1;
    else
        return 0;
}

```

Essa função pode ser re-escrita de forma mais compacta, conforme mostrado abaixo:

```

/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int vazia (Lista* l)
{
    return (l == NULL);
}

```

Função de busca

Outra função útil consiste em verificar se um determinado elemento está presente na lista. A função recebe a informação referente ao elemento que queremos buscar e fornece como valor de retorno o ponteiro do nó da lista que representa o elemento. Caso o elemento não seja encontrado na lista, o valor retornado é NULL.

```

/* função busca: busca um elemento na lista */
Lista* busca (Lista* l, int v)
{
    Lista* p;
    for (p=l; p!=NULL; p=p->prox)
        if (p->info == v)
            return p;
    return NULL;      /* não achou o elemento */
}

```

Função que retira um elemento da lista

Para completar o conjunto de funções que manipulam uma lista, devemos implementar uma função que nos permita retirar um elemento. A função tem como parâmetros de entrada a lista e o valor do elemento que desejamos retirar, e deve retornar o valor atualizado da lista, pois, se o elemento removido for o primeiro da lista, o valor da lista deve ser atualizado.

A função para retirar um elemento da lista é mais complexa. Se descobrirmos que o elemento a ser retirado é o primeiro da lista, devemos fazer com que o novo valor da lista passe a ser o ponteiro para o segundo elemento, e então podemos liberar o espaço alocado para o elemento que queremos retirar. Se o elemento a ser removido estiver no meio da lista, devemos fazer com que o elemento anterior a ele passe a apontar para o elemento seguinte, e então podemos liberar o elemento que queremos retirar. Devemos notar que, no segundo caso, precisamos do ponteiro para o elemento anterior para podermos acertar o encadeamento da lista. As Figuras 9.4 e 9.5 ilustram as operações de remoção.

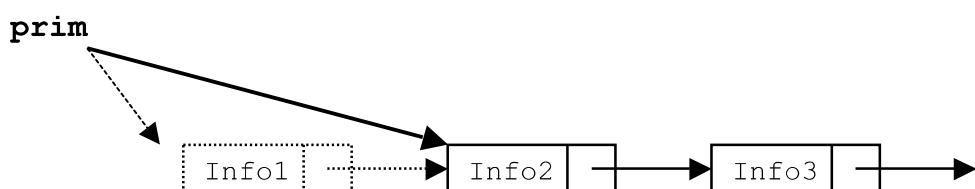


Figura 9.4: Remoção do primeiro elemento da lista.

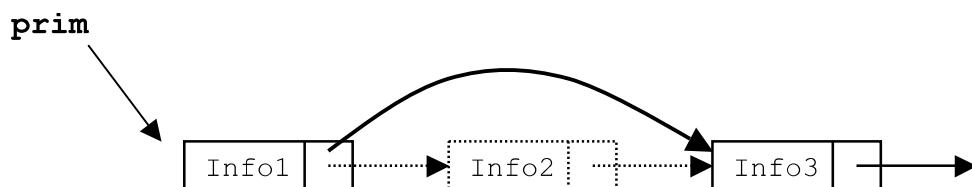


Figura 9.5: Remoção de um elemento no meio da lista.

Uma possível implementação da função para retirar um elemento da lista é mostrada a seguir. Inicialmente, busca-se o elemento que se deseja retirar, guardando uma referência para o elemento anterior.

```

/* função retira: retira elemento da lista */
Lista* retira (Lista* l, int v) {
    Lista* ant = NULL; /* ponteiro para elemento anterior */
    Lista* p = l;       /* ponteiro para percorrer a lista*/

    /* procura elemento na lista, guardando anterior */
    while (p != NULL && p->info != v) {
        ant = p;
        p = p->prox;
    }

    /* verifica se achou elemento */
    if (p == NULL)
        return l; /* não achou: retorna lista original */

    /* retira elemento */
    if (ant == NULL) {
        /* retira elemento do inicio */
        l = p->prox;
    }
    else {
        /* retira elemento do meio da lista */
        ant->prox = p->prox;
    }
    free(p);
    return l;
}

```

O caso de retirar o último elemento da lista recai no caso de retirar um elemento no meio da lista, conforme pode ser observado na implementação acima. Mais adiante, estudaremos a implementação de filas com listas encadeadas. Numa fila, devemos armazenar, além do ponteiro para o primeiro elemento, um ponteiro para o último elemento. Nesse caso, se for removido o último elemento, veremos que será necessário atualizar a fila.

Função para liberar a lista

Uma outra função útil que devemos considerar destrói a lista, liberando todos os elementos alocados. Uma implementação dessa função é mostrada abaixo. A função percorre elemento a elemento, liberando-os. É importante observar que devemos guardar a referência para o próximo elemento antes de liberar o elemento corrente (se liberássemos o elemento e depois tentássemos acessar o encadeamento, estaríamos acessando um espaço de memória que não estaria mais reservado para nosso uso).

```

void libera (Lista* l)
{
    Lista* p = l;
    while (p != NULL) {
        Lista* t = p->prox; /* guarda referência para o próximo elemento
        */
        free(p);           /* libera a memória apontada por p */
        p = t;             /* faz p apontar para o próximo */
    }
}

```

Um programa que ilustra a utilização dessas funções é mostrado a seguir.

```
#include <stdio.h>

int main (void) {
    Lista* l;           /* declara uma lista não iniciada */
    l = inicializa();   /* inicia lista vazia */
    l = insere(l, 23);  /* insere na lista o elemento 23 */
    l = insere(l, 45);  /* insere na lista o elemento 45 */
    l = insere(l, 56);  /* insere na lista o elemento 56 */
    l = insere(l, 78);  /* insere na lista o elemento 78 */
    imprime(l);         /* imprimirá: 78 56 45 23 */
    l = retira(l, 78);  /* imprimirá: 56 45 23 */
    l = retira(l, 45);  /* imprimirá: 56 23 */
    libera(l);
    return 0;
}
```

Mais uma vez, observe que não podemos deixar de atualizar a variável que representa a lista a cada inserção e a cada remoção de um elemento. Esquecer de atribuir o valor de retorno à variável que representa a lista pode gerar erros graves. Se, por exemplo, a função retirar o primeiro elemento da lista, a variável que representa a lista, se não fosse atualizada, estaria apontando para um nó já liberado. Como alternativa, poderíamos fazer com que as funções `insere` e `retira` recebessem o endereço da variável que representa a lista. Nesse caso, os parâmetros das funções seriam do tipo ponteiro para lista (`Lista** l`) e seu conteúdo poderia ser acessado/atualizado de dentro da função usando o operador conteúdo (`*l`).

Manutenção da lista ordenada

A função de inserção vista acima armazena os elementos na lista na ordem inversa à ordem de inserção, pois um novo elemento é sempre inserido no início da lista. Se quisermos manter os elementos na lista numa determinada ordem, temos que encontrar a posição correta para inserir o novo elemento. Essa função não é eficiente, pois temos que percorrer a lista, elemento por elemento, para acharmos a posição de inserção. Se a ordem de armazenamento dos elementos dentro da lista não for relevante, optamos por fazer inserções no início, pois o custo computacional disso independe do número de elementos na lista.

No entanto, se desejarmos manter os elementos em ordem, cada novo elemento deve ser inserido na ordem correta. Para exemplificar, vamos considerar que queremos manter nossa lista de números inteiros em ordem crescente. A função de inserção, neste caso, tem a mesma assinatura da função de inserção mostrada, mas percorre os elementos da lista a fim de encontrar a posição correta para a inserção do novo. Com isto, temos que saber inserir um elemento no meio da lista. A Figura 9.6 ilustra a inserção de um elemento no meio da lista.

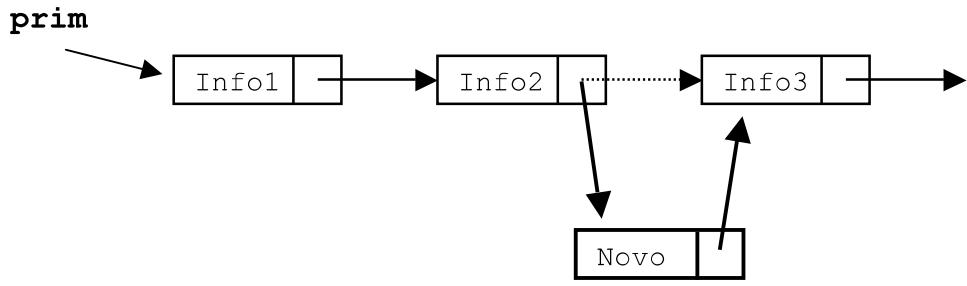


Figura 9.6: Inserção de um elemento no meio da lista.

Conforme ilustrado na figura, devemos localizar o elemento da lista que irá preceder o elemento novo a ser inserido. De posse do ponteiro para esse elemento, podemos encadear o novo elemento na lista. O novo apontará para o próximo elemento na lista e o elemento precedente apontará para o novo. O código abaixo ilustra a implementação dessa função. Neste caso, utilizamos uma função auxiliar responsável por alocar memória para o novo nó e atribuir o campo da informação.

```

/* função auxiliar: cria e inicializa um nó */
Lista* cria (int v)
{
    Lista* p = (Lista*) malloc(sizeof(Lista));
    p->info = v;
    return p;
}

/* função insere_ordenado: insere elemento em ordem */
Lista* insere_ordenado (Lista* l, int v)
{
    Lista* novo = cria(v); /* cria novo nó */
    Lista* ant = NULL; /* ponteiro para elemento anterior */
    Lista* p = l; /* ponteiro para percorrer a lista*/

    /* procura posição de inserção */
    while (p != NULL && p->info < v) {
        ant = p;
        p = p->prox;
    }

    /* insere elemento */
    if (ant == NULL) { /* insere elemento no início */
        novo->prox = l;
        l = novo;
    }
    else { /* insere elemento no meio da lista */
        novo->prox = ant->prox;
        ant->prox = novo;
    }
    return l;
}

```

Devemos notar que essa função, analogamente ao observado para a função de remoção, também funciona se o elemento tiver que ser inserido no final da lista.

10.2. Implementações recursivas

Uma lista pode ser definida de maneira recursiva. Podemos dizer que uma lista encadeada é representada por:

- uma lista vazia; ou
- um elemento seguido de uma (sub-)lista.

Neste caso, o segundo elemento da lista representa o primeiro elemento da sub-lista. Com base na definição recursiva, podemos implementar as funções de lista recursivamente. Por exemplo, a função para imprimir os elementos da lista pode ser reescrita da forma ilustrada abaixo:

```
/* Função imprime recursiva */
void imprime_rec (Lista* l)
{
    if (vazia(l))
        return;
    /* imprime primeiro elemento */
    printf("info: %d\n", l->info);
    /* imprime sub-lista */
    imprime_rec(l->prox);
}
```

É fácil alterarmos o código acima para obtermos a impressão dos elementos da lista em ordem inversa: basta invertermos a ordem das chamadas às funções `printf` e `imprime_rec`.

A função para retirar um elemento da lista também pode ser escrita de forma recursiva. Neste caso, só retiramos um elemento se ele for o primeiro da lista (ou da sub-lista). Se o elemento que queremos retirar não for o primeiro, chamamos a função recursivamente para retirar o elemento da sub-lista.

```
/* Função retira recursiva */
Lista* retira_rec (Lista* l, int v)
{
    if (vazia(l))
        return l; /* lista vazia: retorna valor original */

    /* verifica se elemento a ser retirado é o primeiro */
    if (l->info == v) {
        Lista* t = l; /* temporário para poder liberar */
        l = l->prox;
        free(t);
    }
    else {
        /* retira de sub-lista */
        l->prox = retira_rec(l->prox, v);
    }
    return l;
}
```

A função para liberar uma lista também pode ser escrita recursivamente, de forma bastante simples. Nessa função, se a lista não for vazia, liberamos primeiro a sub-lista e depois liberamos a lista.

```

void libera_rec (Lista* l)
{
    if (!vazia(l))
    {
        libera_rec(l->prox);
        free(l);
    }
}

```

Exercício: Implemente uma função que verifique se duas listas encadeadas são iguais. Duas listas são consideradas iguais se têm a mesma seqüência de elementos. O protótipo da função deve ser dado por:

```
int igual (Lista* l1, Lista* l2);
```

Exercício: Implemente uma função que crie uma cópia de uma lista encadeada. O protótipo da função deve ser dado por:

```
List* copia (List* l);
```

10.3. Listas genéricas

Um nó de uma lista encadeada contém basicamente duas informações: o encadeamento e a informação armazenada. Assim, a estrutura de um nó para representar uma lista de números inteiros é dada por:

```

struct lista {
    int info;
    struct lista *prox;
};
typedef struct lista List;

```

Analogamente, se quisermos representar uma lista de números reais, podemos definir a estrutura do nó como sendo:

```

struct lista {
    float info;
    struct lista *prox;
};
typedef struct lista List;

```

A informação armazenada na lista não precisa ser necessariamente um dado simples. Podemos, por exemplo, considerar a construção de uma lista para armazenar um conjunto de retângulos. Cada retângulo é definido pela base *b* e pela altura *h*. Assim, a estrutura do nó pode ser dada por:

```

struct lista {
    float b;
    float h;
    struct lista *prox;
};
typedef struct lista List;

```

Esta mesma composição pode ser escrita de forma mais clara se definirmos um tipo adicional que represente a informação. Podemos definir um tipo *Retangulo* e usá-lo para representar a informação armazenada na lista.

```

struct retangulo {
    float b;
    float h;
};
typedef struct retangulo Retangulo;

```

```

struct lista {
    Retangulo info;
    struct lista *prox;
};
typedef struct lista Lista;

```

Aqui, a informação volta a ser representada por um único campo (`info`), que é uma estrutura. Se `p` fosse um ponteiro para um nó da lista, o valor da base do retângulo armazenado nesse nó seria acessado por: `p->info.b`.

Ainda mais interessante é termos o campo da informação representado por um ponteiro para a estrutura, em vez da estrutura em si.

```

struct retangulo {
    float b;
    float h;
};
typedef struct retangulo Retangulo;

struct lista {
    Retangulo *info;
    struct lista *prox;
};
typedef struct lista Lista;

```

Neste caso, para criarmos um nó, temos que fazer duas alocações dinâmicas: uma para criar a estrutura do retângulo e outra para criar a estrutura do nó. O código abaixo ilustra uma função para a criação de um nó.

```

Lista* cria (void)
{
    Retangulo* r = (Retangulo*) malloc(sizeof(Retangulo));
    Lista* p = (Lista*) malloc(sizeof(Lista));
    p->info = r;
    p->prox = NULL;
    return p;
}

```

Naturalmente, o valor da base associado a um nó `p` seria agora acessado por: `p->info->b`. A vantagem dessa representação (utilizando ponteiros) é que, independente da informação armazenada na lista, a estrutura do nó é sempre composta por um ponteiro para a informação e um ponteiro para o próximo nó da lista.

A representação da informação por um ponteiro nos permite construir listas heterogêneas, isto é, listas em que as informações armazenadas diferem de nó para nó. Diversas aplicações precisam construir listas heterogêneas, pois necessitam agrupar elementos afins mas não necessariamente iguais. Como exemplo, vamos considerar uma aplicação que necessite manipular listas de objetos geométricos planos para cálculos de áreas. Para simplificar, vamos considerar que os objetos podem ser apenas retângulos, triângulos ou círculos. Sabemos que as áreas desses objetos são dadas por:

$$r = b * h \quad t = \frac{b * h}{2} \quad c = \pi r^2$$

Devemos definir um tipo para cada objeto a ser representado:

```
struct retangulo {
    float b;
    float h;
};

typedef struct retangulo Retangulo;

struct triangulo {
    float b;
    float h;
};

typedef struct triangulo Triangulo;

struct circulo {
    float r;
};

typedef struct circulo Circulo;
```

O nó da lista deve ser composto por três campos:

- um identificador de qual objeto está armazenado no nó
- um ponteiro para a estrutura que contém a informação
- um ponteiro para o próximo nó da lista

É importante salientar que, a rigor, a lista é homogênea, no sentido de que todos os nós contêm as mesmas informações. O ponteiro para a informação deve ser do tipo genérico, pois não sabemos a princípio para que estrutura ele irá apontar: pode apontar para um retângulo, um triângulo ou um círculo. Um ponteiro genérico em C é representado pelo tipo `void*`. A função do tipo “ponteiro genérico” pode representar qualquer endereço de memória, independente da informação de fato armazenada nesse espaço. No entanto, de posse de um ponteiro genérico, não podemos acessar a memória por ele apontada, já que não sabemos a informação armazenada. Por esta razão, o nó de uma lista genérica deve guardar explicitamente um identificador do tipo de objeto de fato armazenado. Consultando esse identificador, podemos converter o ponteiro genérico no ponteiro específico para o objeto em questão e, então, acessarmos os campos do objeto.

Como identificador de tipo, podemos usar valores inteiros definidos como constantes simbólicas:

```
#define RET 0
#define TRI 1
#define CIR 2
```

Assim, na criação do nó, armazenamos o identificador de tipo correspondente ao objeto sendo representado. A estrutura que representa o nó pode ser dada por:

```
/* Define o nó da estrutura */
struct listagen {
    int tipo;
    void *info;
    struct listagen *prox;
};
typedef struct listagen ListaGen;
```

A função para a criação de um nó da lista pode ser definida por três variações, uma para cada tipo de objeto que pode ser armazenado.

```

/* Cria um nó com um retângulo, inicializando os campos base e altura
*/
ListaGen* cria_ret (float b, float h)
{
    Retangulo* r;
    ListaGen* p;

    /* aloca retângulo */
    r = (Retangulo*) malloc(sizeof(Retangulo));
    r->b = b;
    r->h = h;

    /* aloca nó */
    p = (ListaGen*) malloc(sizeof(ListaGen));
    p->tipo = RET;
    p->info = r;
    p->prox = NULL;

    return p;
}

/* Cria um nó com um triângulo, inicializando os campos base e altura
*/
ListaGen* cria_tri (float b, float h)
{
    Triangulo* t;
    ListaGen* p;

    /* aloca triângulo */
    t = (Triangulo*) malloc(sizeof(Triangulo));
    t->b = b;
    t->h = h;

    /* aloca nó */
    p = (ListaGen*) malloc(sizeof(ListaGen));
    p->tipo = TRI;
    p->info = t;

    p->prox = NULL;
    return p;
}

/* Cria um nó com um círculo, inicializando o campo raio */
ListaGen* cria_cir (float r)
{
    Circulo* c;
    ListaGen* p;

    /* aloca círculo */
    c = (Circulo*) malloc(sizeof(Circulo));
    c->r = r;

    /* aloca nó */
    p = (ListaGen*) malloc(sizeof(ListaGen));
    p->tipo = CIR;
    p->info = c;
    p->prox = NULL;

    return p;
}

```

Uma vez criado o nó, podemos inseri-lo na lista como já vínhamos fazendo com nós de listas homogêneas. As constantes simbólicas que representam os tipos dos objetos podem ser agrupadas numa enumeração (ver seção 7.5):

```

enum {
    RET,
    TRI,
    CIR
};
```

Manipulação de listas heterogêneas

Para exemplificar a manipulação de listas heterogêneas, considerando a existência de uma lista com os objetos geométricos apresentados acima, vamos implementar uma função que forneça como valor de retorno a maior área entre os elementos da lista. Uma implementação dessa função é mostrada abaixo, onde criamos uma função auxiliar que calcula a área do objeto armazenado num determinado nó da lista:

```

#define PI 3.14159

/* função auxiliar: calcula área correspondente ao nó */
float area (ListaGen *p)
{
    float a;           /* área do elemento */

    switch (p->tipo) {

        case RET:
        {
            /* converte para retângulo e calcula área */
            Retangulo *r = (Retangulo*) p->info;
            a = r->b * r->h;
        }
        break;

        case TRI:
        {
            /* converte para triângulo e calcula área */
            Triangulo *t = (Triangulo*) p->info;
            a = (t->b * t->h) / 2;
        }
        break;

        case CIR:
        {
            /* converte para círculo e calcula área */
            Circulo *c = (Circulo)p->info;
            a = PI * c->r * c->r;
        }
        break;
    }
    return a;
}

/* Função para cálculo da maior área */
float max_area (ListaGen* l)
{
    float amax = 0.0;      /* maior área */
    ListaGen* p;
    for (p=l; p!=NULL; p=p->prox) {
        float a = area(p);      /* área do nó */
        if (a > amax)
            amax = a;
    }
    return amax;
}
```

A função para o cálculo da área mostrada acima pode ser subdivida em funções específicas para o cálculo das áreas de cada objeto geométrico, resultando em um código mais estruturado.

```
/* função para cálculo da área de um retângulo */
float ret_area (Retangulo* r)
{
    return r->b * r->h;
}

/* função para cálculo da área de um triângulo */
float tri_area (Triangulo* t)
{
    return (t->b * t->h) / 2;
}

/* função para cálculo da área de um círculo */
float cir_area (Circulo* c)
{
    return PI * c->r * c->r;
}

/* função para cálculo da área do nó (versão 2) */
float area (ListaGen* p)
{
    float a;
    switch (p->tipo) {
        case RET:
            a = ret_area(p->info);
            break;
        case TRI:
            a = tri_area(p->info);
            break;
        case CIR:
            a = cir_area(p->info);
            break;
    }
    return a;
}
```

Neste caso, a conversão de ponteiro genérico para ponteiro específico é feita quando chamamos uma das funções de cálculo da área: passa-se um ponteiro genérico que é atribuído, através da conversão implícita de tipo, para um ponteiro específico¹.

Devemos salientar que, quando trabalhamos com conversão de ponteiros genéricos, temos que garantir que o ponteiro armazene o endereço onde de fato existe o tipo específico correspondente. O compilador não tem como checar se a conversão é válida; a verificação do tipo passa a ser responsabilidade do programador.

10.4. Listas circulares

Algumas aplicações necessitam representar conjuntos cílicos. Por exemplo, as arestas que delimitam uma face podem ser agrupadas por uma estrutura circular. Para esses casos, podemos usar *listas circulares*.

Numa lista circular, o último elemento tem como próximo o primeiro elemento da lista, formando um ciclo. A rigor, neste caso, não faz sentido falarmos em primeiro ou último

¹ Este código não é válido em C++. A linguagem C++ não tem conversão implícita de um ponteiro genérico para um ponteiro específico. Para compilar em C++, devemos fazer a conversão explicitamente. Por exemplo:

```
a = ret_area((Retangulo*)p->info);
```

elemento. A lista pode ser representada por um ponteiro para um elemento inicial qualquer da lista. A Figura 9.7 ilustra o arranjo da memória para a representação de uma lista circular.

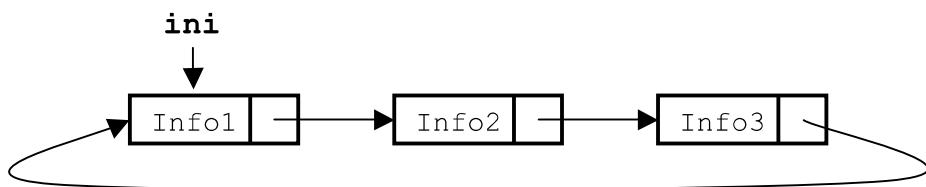


Figura 9.7: Arranjo da memória de uma lista circular.

Para percorrer os elementos de uma lista circular, visitamos todos os elementos a partir do ponteiro do elemento inicial até alcançarmos novamente esse mesmo elemento. O código abaixo exemplifica essa forma de percorrer os elementos. Neste caso, para simplificar, consideramos uma lista que armazena valores inteiros. Devemos salientar que o caso em que a lista é vazia ainda deve ser tratado (se a lista é vazia, o ponteiro para um elemento inicial vale NULL).

```

void imprime_circular (Lista* l)
{
    Lista* p = l;           /* faz p apontar para o nó inicial */
    /* testa se lista não é vazia */
    if (p) {
    {
        /* percorre os elementos até alcançar novamente o início */
        do {
            printf("%d\n", p->info);    /* imprime informação do nó */
            p = p->prox;                /* avança para o próximo nó */
        } while (p != l);
    }
}
  
```

Exercício: Escreva as funções para inserir e retirar um elemento de uma lista circular.

10.5. Listas duplamente encadeadas**

A estrutura de lista encadeada vista nas seções anteriores caracteriza-se por formar um encadeamento simples entre os elementos: cada elemento armazena um ponteiro para o próximo elemento da lista. Desta forma, não temos como percorrer eficientemente os elementos em ordem inversa, isto é, do final para o início da lista. O encadeamento simples também dificulta a retirada de um elemento da lista. Mesmo se tivermos o ponteiro do elemento que desejamos retirar, temos que percorrer a lista, elemento por elemento, para encontrarmos o elemento anterior, pois, dado um determinado elemento, não temos como acessar diretamente seu elemento anterior.

Para solucionar esses problemas, podemos formar o que chamamos de *listas duplamente encadeadas*. Nelas, cada elemento tem um ponteiro para o próximo elemento e um ponteiro para o elemento anterior. Desta forma, dado um elemento, podemos acessar ambos os elementos adjacentes: o próximo e o anterior. Se tivermos um ponteiro para o último elemento da lista, podemos percorrer a lista em ordem inversa, bastando acessar continuamente o elemento anterior, até alcançar o primeiro elemento da lista, que não tem elemento anterior (o ponteiro do elemento anterior vale NULL).

A Figura 9.8 esquematiza a estruturação de uma lista duplamente encadeada.

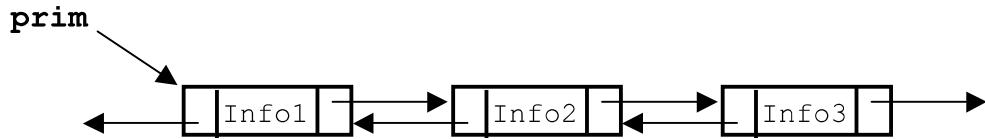


Figura 9.8: Arranjo da memória de uma lista duplamente encadeada.

Para exemplificar a implementação de listas duplamente encadeadas, vamos novamente considerar o exemplo simples no qual queremos armazenar valores inteiros na lista. O nó da lista pode ser representado pela estrutura abaixo e a lista pode ser representada através do ponteiro para o primeiro nó.

```
struct lista2 {
    int info;
    struct lista2* ant;
    struct lista2* prox;
};

typedef struct Lista2 Lista2;
```

Com base nas definições acima, exemplificamos a seguir a implementação de algumas funções que manipulam listas duplamente encadeadas.

Função de inserção

O código a seguir mostra uma possível implementação da função que insere novos elementos no início da lista. Após a alocação do novo elemento, a função acertar o duplo encadeamento.

```
/* inserção no início */
Lista2* insere (Lista2* l, int v)
{
    Lista2* novo = (Lista2*) malloc(sizeof(Lista2));
    novo->info = v;
    novo->prox = l;
    novo->ant = NULL;
    /* verifica se lista não está vazia */
    if (l != NULL)
        l->ant = novo;
    return novo;
}
```

Nessa função, o novo elemento é encadeado no início da lista. Assim, ele tem como próximo elemento o antigo primeiro elemento da lista e como anterior o valor `NULL`. A seguir, a função testa se a lista não era vazia, pois, neste caso, o elemento anterior do então primeiro elemento passa a ser o novo elemento. De qualquer forma, o novo elemento passa a ser o primeiro da lista, e deve ser retornado como valor da lista atualizada. A Figura 9.9 ilustra a operação de inserção de um novo elemento no início da lista.

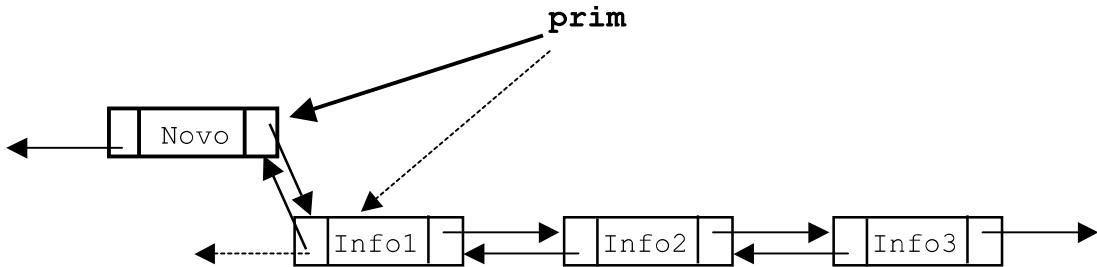


Figura 9.9: Inserção de um novo elemento no início da lista.

Função de busca

A função de busca recebe a informação referente ao elemento que queremos buscar e tem como valor de retorno o ponteiro do nó da lista que representa o elemento. Caso o elemento não seja encontrado na lista, o valor retornado é `NULL`.

```
/* função busca: busca um elemento na lista */
Lista2* busca (Lista2* l, int v)
{
    Lista2* p;
    for (p=l; p!=NULL; p=p->prox)
        if (p->info == v)
            return p;
    return NULL;           /* não achou o elemento */
}
```

Função que retira um elemento da lista

A função de remoção fica mais complicada, pois temos que acertar o encadeamento duplo. Em contrapartida, podemos retirar um elemento da lista conhecendo apenas o ponteiro para esse elemento. Desta forma, podemos usar a função de busca acima para localizar o elemento e em seguida acertar o encadeamento, liberando o elemento ao final.

Se `p` representa o ponteiro do elemento que desejamos retirar, para acertar o encadeamento devemos conceitualmente fazer:

```
p->ant->prox = p->prox;
p->prox->ant = p->ant;
```

Isto é, o anterior passa a apontar para o próximo e o próximo passa a apontar para o anterior. Quando `p` apontar para um elemento no meio da lista, as duas atribuições acima são suficientes para efetivamente acertar o encadeamento da lista. No entanto, se `p` for um elemento no extremo da lista, devemos considerar as condições de contorno. Se `p` for o primeiro, não podemos escrever `p->ant->prox`, pois `p->ant` é `NULL`; além disso, temos que atualizar o valor da lista, pois o primeiro elemento será removido.

Uma implementação da função para retirar um elemento é mostrada a seguir:

```
/* função retira: retira elemento da lista */
Lista2* retira (Lista2* l, int v) {
    Lista2* p = busca(l,v);

    if (p == NULL)
        return l; /* não achou o elemento: retorna lista inalterada */

    /* retira elemento do encadeamento */
    if (l == p)
        l = p->prox;
    else
        p->ant->prox = p->prox;

    if (p->prox != NULL)
        p->prox->ant = p->ant;

    free(p);

    return l;
}
```

Lista circular duplamente encadeada

Uma lista circular também pode ser construída com encadeamento duplo. Neste caso, o que seria o último elemento da lista passa ter como próximo o primeiro elemento, que, por sua vez, passa a ter o último como anterior. Com essa construção podemos percorrer a lista nos dois sentidos, a partir de um ponteiro para um elemento qualquer. Abaixo, ilustramos o código para imprimir a lista no sentido reverso, isto é, percorrendo o encadeamento dos elementos anteriores.

```
void imprime_circular_rev (Lista2* l)
{
    Lista2* p = l; /* faz p apontar para o nó inicial */
    /* testa se lista não é vazia */
    if (p) {
    {
        /* percorre os elementos até alcançar novamente o início */
        do {
            printf("%d\n", p->info); /* imprime informação do nó */
            p = p->ant; /* "avança" para o nó anterior */
        } while (p != l);
    }
}
```

Exercício: Escreva as funções para inserir e retirar um elemento de uma lista circular duplamente encadeada.

11. Pilhas

W. Celes e J. L. Rangel

Uma das estruturas de dados mais simples é a pilha. Possivelmente por essa razão, é a estrutura de dados mais utilizada em programação, sendo inclusive implementada diretamente pelo *hardware* da maioria das máquinas modernas. A idéia fundamental da pilha é que todo o acesso a seus elementos é feito através do seu topo. Assim, quando um elemento novo é introduzido na pilha, passa a ser o elemento do topo, e o único elemento que pode ser removido da pilha é o do topo. Isto faz com que os elementos da pilha sejam retirados na ordem inversa à ordem em que foram introduzidos: o primeiro que sai é o último que entrou (a sigla LIFO – *last in, first out* – é usada para descrever esta estratégia).

Para entendermos o funcionamento de uma estrutura de pilha, podemos fazer uma analogia com uma pilha de pratos. Se quisermos adicionar um prato na pilha, o colocamos no topo. Para pegar um prato da pilha, retiramos o do topo. Assim, temos que retirar o prato do topo para ter acesso ao próximo prato. A estrutura de pilha funciona de maneira análoga. Cada novo elemento é inserido no topo e só temos acesso ao elemento do topo da pilha.

Existem duas operações básicas que devem ser implementadas numa estrutura de pilha: a operação para empilhar um novo elemento, inserindo-o no topo, e a operação para desempilhar um elemento, removendo-o do topo. É comum nos referirmos a essas duas operações pelos termos em inglês *push* (empilhar) e *pop* (desempilhar). A Figura 10.1 ilustra o funcionamento conceitual de uma pilha.

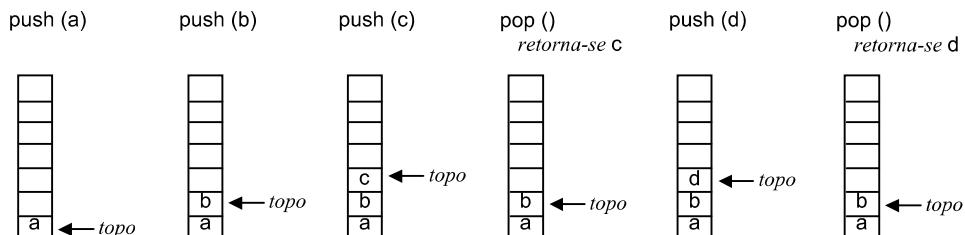


Figura 10.1: Funcionamento da pilha.

O exemplo de utilização de pilha mais próximo é a própria pilha de execução da linguagem C. As variáveis locais das funções são dispostas numa pilha e uma função só tem acesso às variáveis que estão no topo (não é possível acessar as variáveis da função locais às outras funções).

Há várias implementações possíveis de uma pilha, que se distinguem pela natureza dos seus elementos, pela maneira como os elementos são armazenados e pelas operações disponíveis para o tratamento da pilha.

11.1. Interface do tipo pilha

Neste capítulo, consideraremos duas implementações de pilha: usando vetor e usando lista encadeada. Para simplificar a exposição, consideraremos uma pilha que armazena valores reais. Independente da estratégia de implementação, podemos definir a interface do tipo abstrato que representa uma estrutura de pilha. A interface é composta pelas operações que estarão disponibilizadas para manipular e acessar as informações da pilha. Neste exemplo, vamos considerar a implementação de cinco operações:

- criar uma estrutura de pilha;
- inserir um elemento no topo (*push*);
- remover o elemento do topo (*pop*);
- verificar se a pilha está vazia;
- liberar a estrutura de pilha.

O arquivo `pilha.h`, que representa a interface do tipo, pode conter o seguinte código:

```
typedef struct pilha Pilha;  
  
Pilha* cria (void);  
void push (Pilha* p, float v);  
float pop (Pilha* p);  
int vazia (Pilha* p);  
void libera (Pilha* p);
```

A função `cria` aloca dinamicamente a estrutura da pilha, inicializa seus campos e retorna seu ponteiro; as funções `push` e `pop` inserem e removem, respectivamente, um valor real na pilha; a função `vazia` informa se a pilha está ou não vazia; e a função `libera` destrói a pilha, liberando toda a memória usada pela estrutura.

11.2. Implementação de pilha com vetor

Em aplicações computacionais que precisam de uma estrutura de pilha, é comum sabermos de antemão o número máximo de elementos que podem estar armazenados simultaneamente na pilha, isto é, a estrutura da pilha tem um limite conhecido. Nestes casos, a implementação da pilha pode ser feita usando um vetor. A implementação com vetor é bastante simples. Devemos ter um vetor (`vet`) para armazenar os elementos da pilha. Os elementos inseridos ocupam as primeiras posições do vetor. Desta forma, se temos `n` elementos armazenados na pilha, o elemento `vet[n-1]` representa o elemento do topo.

A estrutura que representa o tipo pilha deve, portanto, ser composta pelo vetor e pelo número de elementos armazenados.

```
#define MAX 50  
  
struct pilha {  
    int n;  
    float vet[MAX];  
};
```

A função para criar a pilha aloca dinamicamente essa estrutura e inicializa a pilha como sendo vazia, isto é, com o número de elementos igual a zero.

```
Pilha* cria (void)
{
    Pilha* p = (Pilha*) malloc(sizeof(Pilha));
    p->n = 0; /* inicializa com zero elementos */
    return p;
}
```

Para inserir um elemento na pilha, usamos a próxima posição livre do vetor. Devemos ainda assegurar que exista espaço para a inserção do novo elemento, tendo em vista que trata-se de um vetor com dimensão fixa.

```
void push (Pilha* p, float v)
{
    if (p->n == MAX) { /* capacidade esgotada */
        printf("Capacidade da pilha estourou.\n");
        exit(1); /* aborta programa */
    }
    /* insere elemento na próxima posição livre */
    p->vet[p->n] = v;
    p->n++;
}
```

A função `pop` retira o elemento do topo da pilha, fornecendo seu valor como retorno. Podemos também verificar se a pilha está ou não vazia.

```
float pop (Pilha* p)
{
    float v;
    if (vazia(p)) {
        printf("Pilha vazia.\n");
        exit(1); /* aborta programa */
    }
    /* retira elemento do topo */
    v = p->vet[p->n-1];
    p->n--;
    return v;
}
```

A função que verifica se a pilha está vazia pode ser dada por:

```
int vazia (Pilha* p)
{
    return (p->n == 0);
}
```

Finalmente, a função para liberar a memória alocada pela pilha pode ser:

```
void libera (Pilha* p)
{
    free(p);
}
```

11.3. Implementação de pilha com lista

Quando o número máximo de elementos que serão armazenados na pilha não é conhecido, devemos implementar a pilha usando uma estrutura de dados dinâmica, no caso, empregando uma lista encadeada. Os elementos são armazenados na lista e a pilha pode ser representada simplesmente por um ponteiro para o primeiro nó da lista.

O nó da lista para armazenar valores reais pode ser dado por:

```
struct no {
    float info;
    struct no* prox;
};
typedef struct no No;
```

A estrutura da pilha é então simplesmente:

```
struct pilha {
    No* prim;
};
```

A função `cria` aloca a estrutura da pilha e inicializa a lista como sendo vazia.

```
Pilha* cria (void)
{
    Pilha* p = (Pilha*) malloc(sizeof(Pilha));
    p->prim = NULL;
    return p;
}
```

O primeiro elemento da lista representa o topo da pilha. Cada novo elemento é inserido no início da lista e, consequentemente, sempre que solicitado, retiramos o elemento também do início da lista. Desta forma, precisamos de duas funções auxiliares da lista: para inserir no início e para remover do início. Ambas as funções retornam o novo primeiro nó da lista.

```
/* função auxiliar: insere no inicio */
No* ins_ini (No* l, float v)
{
    No* p = (No*) malloc(sizeof(No));
    p->info = v;
    p->prox = l;
    return p;
}

/* função auxiliar: retira do inicio */
No* ret_ini (No* l)
{
    No* p = l->prox;
    free(l);
    return p;
}
```

As funções que manipulam a pilha fazem uso dessas funções de lista:

```
void push (Pilha* p, float v)
{
    p->prim = ins_ini(p->prim,v);
}

float pop (Pilha* p)
{
    float v;
    if (vazia(p)) {
        printf("Pilha vazia.\n");
        exit(1); /* aborta programa */
    }
    v = p->prim->info;
    p->prim = ret_ini(p->prim);
    return v;
}
```

A pilha estará vazia se a lista estiver vazia:

```
int vazia (Pilha* p)
{
    return (p->prim==NULL);
}
```

Por fim, a função que libera a pilha deve antes liberar todos os elementos da lista.

```
void libera (Pilha* p)
{
    No* q = p->prim;
    while (q!=NULL) {
        No* t = q->prox;
        free(q);
        q = t;
    }
    free(p);
}
```

A rigor, pela definição da estrutura de pilha, só temos acesso ao elemento do topo. No entanto, para testar o código, pode ser útil implementarmos uma função que imprima os valores armazenados na pilha. Os códigos abaixo ilustram a implementação dessa função nas duas versões de pilha (vetor e lista). A ordem de impressão adotada é do topo para a base.

```
/* imprime: versão com vetor */
void imprime (Pilha* p)
{
    int i;
    for (i=p->n-1; i>=0; i--)
        printf("%f\n",p->vet[i]);
}

/* imprime: versão com lista */
void imprime (Pilha* p)
{
    No* q;
    for (q=p->prim; q!=NULL; q=q->prox)
        printf("%f\n",q->info);
}
```

11.4. Exemplo de uso: calculadora pós-fixada

Um bom exemplo de aplicação de pilha é o funcionamento das calculadoras da HP (Hewlett-Packard). Elas trabalham com expressões pós-fixadas, então para avaliarmos uma expressão como $(1-2)*(4+5)$ podemos digitar $1\ 2\ -\ 4\ 5\ +\ *$. O funcionamento dessas calculadoras é muito simples. Cada operando é empilhado numa pilha de valores. Quando se encontra um operador, desempilha-se o número apropriado de operandos (dois para operadores binários e um para operadores unários), realiza-se a operação devida e empilha-se o resultado. Deste modo, na expressão acima, são empilhados os valores 1 e 2. Quando aparece o operador $-$, 1 e 2 são desempilhados e o resultado da operação, no caso $-1 (= 1 - 2)$, é colocado no topo da pilha. A seguir, 4 e 5 são empilhados. O operador seguinte, $+$, desempilha o 4 e o 5 e empilha o resultado da soma, 9. Nesta hora, estão na pilha os dois resultados parciais, -1 na base e 9 no topo. O operador $*$, então, desempilha os dois e coloca $-9 (= -1 * 9)$ no topo da pilha.

Como exemplo de aplicação de uma estrutura de pilha, vamos implementar uma calculadora pós-fixada. Ela deve ter uma pilha de valores reais para representar os operandos. Para enriquecer a implementação, vamos considerar que o formato com que os valores da pilha são impressos seja um dado adicional associado à calculadora. Esse formato pode, por exemplo, ser passado quando da criação da calculadora.

Para representar a interface exportada pela calculadora, podemos criar o arquivo `calc.h`:

```
/* Arquivo que define a interface da calculadora */

typedef struct calc Calc;

/* funções exportadas */
Calc* cria_calc (char* f);
void operando (Calc* c, float v);
void operador (Calc* c, char op);
void libera_calc (Calc* c);
```

Essas funções utilizam as funções mostradas acima, independente da implementação usada na pilha (vetor ou lista). O tipo que representa a calculadora pode ser dado por:

```
struct calc {
    char f[21];      /* formato para impressão */
    Pilha* p;        /* pilha de operandos */
};
```

A função `cria` recebe como parâmetro de entrada uma cadeia de caracteres com o formato que será utilizado pela calculadora para imprimir os valores. Essa função cria uma calculadora inicialmente sem operandos na pilha.

```
Calc* cria_calc (char* formato)
{
    Calc* c = (Calc*) malloc(sizeof(Calc));
    strcpy(c->f, formato);
    c->p = cria();           /* cria pilha vazia */
    return c;
}
```

A função `operando` coloca no topo da pilha o valor passado como parâmetro. A função `operador` retira os dois valores do topo da pilha (só consideraremos operadores binários), efetua a operação correspondente e coloca o resultado no topo da pilha. As operações válidas são: '+' para somar, '-' para subtrair, '*' para multiplicar e '/' para dividir. Se não existirem operandos na pilha, consideraremos que seus valores são zero. Tanto a função `operando` quanto a função `operador` imprimem, utilizando o formato especificado na função `cria`, o novo valor do topo da pilha.

```
void operando (Calc* c, float v)
{
    /* empilha operando */
    push(c->p, v);

    /* imprime topo da pilha */
    printf(c->f, v);
}
```

```

void operador (Calc* c, char op)
{
    float v1, v2, v;

    /* desempilha operandos */
    if (vazia(c->p))
        v2 = 0.0;
    else
        v2 = pop(c->p);
    if (vazia(c->p))
        v1 = 0.0;
    else
        v1 = pop(c->p);

    /* faz operação */
    switch (op) {
        case '+': v = v1+v2; break;
        case '-': v = v1-v2; break;
        case '*': v = v1*v2; break;
        case '/': v = v1/v2; break;
    }

    /* empilha resultado */
    push(c->p,v);

    /* imprime topo da pilha */
    printf(c->f,v);
}

```

Por fim, a função para liberar a memória usada pela calculadora libera a pilha de operandos e a estrutura da calculadora.

```

void libera_calc (Calc* c)
{
    libera(c->p);
    free(c);
}

```

Um programa cliente que faça uso da calculadora é mostrado abaixo:

```

/* Programa para ler expressão e chamar funções da calculadora */

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

int main (void)
{
    char c;
    float v;
    Calc* calc;

    /* cria calculadora com precisão de impressão de duas casas decimais */
    calc = cria_calc("%.2f\n");

    do {
        /* le proximo caractere não branco */
        scanf(" %c",&c);
        /* verifica se é operador válido */
        if (c=='+' || c=='-' || c=='*' || c=='/') {
            operador(calc,c);
        }
        /* devolve caractere lido e tenta ler número */
        else {
            ungetc(c,stdin);
            if (scanf("%f",&v) == 1)
                operando(calc,v);
        }
    }
}

```

```

        }
    } while (c != 'q');
libera_calc(calc);
return 0;
}

```

Esse programa cliente lê os dados fornecidos pelo usuário e opera a calculadora. Para tanto, o programa lê um caractere e verifica se é um operador válido. Em caso negativo, o programa “devolve” o caractere lido para o *buffer* de leitura, através da função ungetc, e tenta ler um operando. O usuário finaliza a execução do programa digitando q.

Se executado, e considerando-se as expressões digitadas pelo usuário mostradas abaixo, esse programa teria como saída:

3 5 8 * + 3.00 5.00 8.00 40.00 43.00 7 / 7.00 6.14 q	<small>← digitado pelo usuário</small> <small>← digitado pelo usuário</small> <small>← digitado pelo usuário</small>
--	--

Exercício: Estenda a funcionalidade da calculadora incluindo novos operadores unários e binários (sugestão: ~ como menos unário, # como raiz quadrada, ^ como exponenciação).

12. Filas

W. Celes e J. L. Rangel

Outra estrutura de dados bastante usada em computação é a *fila*. Na estrutura de fila, os acessos aos elementos também seguem uma regra. O que diferencia a *fila* da *pilha* é a ordem de saída dos elementos: enquanto na pilha “o último que entra é o primeiro que sai”, na fila “o primeiro que entra é o primeiro que sai” (a sigla FIFO – *first in, first out* – é usada para descrever essa estratégia). A idéia fundamental da fila é que só podemos inserir um novo elemento no final da fila e só podemos retirar o elemento do início.

A estrutura de fila é uma analogia natural com o conceito de fila que usamos no nosso dia a dia: quem primeiro entra numa fila é o primeiro a ser atendido (a sair da fila). Um exemplo de utilização em computação é a implementação de uma fila de impressão. Se uma impressora é compartilhada por várias máquinas, deve-se adotar uma estratégia para determinar que documento será impresso primeiro. A estratégia mais simples é tratar todas as requisições com a mesma prioridade e imprimir os documentos na ordem em que foram submetidos – o primeiro submetido é o primeiro a ser impresso.

De modo análogo ao que fizemos com a estrutura de pilha, neste capítulo discutiremos duas estratégias para a implementação de uma estrutura de fila: usando vetor e usando lista encadeada. Para implementar uma fila, devemos ser capazes de inserir novos elementos em uma extremidade, o *fim*, e retirar elementos da outra extremidade, o *início*.

12.1. Interface do tipo fila

Antes de discutirmos as duas estratégias de implementação, podemos definir a interface disponibilizada pela estrutura, isto é, definir quais operações serão implementadas para manipular a fila. Mais uma vez, para simplificar a exposição, consideraremos uma estrutura que armazena valores reais. Independente da estratégia de implementação, a interface do tipo abstrato que representa uma estrutura de fila pode ser composta pelas seguintes operações:

- criar uma estrutura de fila;
- inserir um elemento no fim;
- retirar o elemento do início;
- verificar se a fila está vazia;
- liberar a fila.

O arquivo *fila.h*, que representa a interface do tipo, pode conter o seguinte código:

```
typedef struct fila Fila;

Fila* cria (void);
void insere (Fila* f, float v);
float retira (Fila* f);
int vazia (Fila* f);
void libera (Fila* f);
```

A função *cria* aloca dinamicamente a estrutura da fila, inicializa seus campos e retorna seu ponteiro; a função *insere* adiciona um novo elemento no final da fila e a função

`remove` remove o elemento do início; a função `vazia` informa se a fila está ou não vazia; e a função `libera` destrói a estrutura, liberando toda a memória alocada.

12.2. Implementação de fila com vetor

Como no caso da pilha, nossa primeira implementação de fila será feita usando um vetor para armazenar os elementos. Para isso, devemos fixar o número máximo N de elementos na fila. Podemos observar que o processo de inserção e remoção em extremidades opostas fará com que a fila “ande” no vetor. Por exemplo, se inserirmos os elementos 1.4, 2.2, 3.5, 4.0 e depois retirarmos dois elementos, a fila não estará mais nas posições iniciais do vetor. A Figura 11.1 ilustra a configuração da fila após a inserção dos primeiros quatro elementos e a Figura 11.2 após a remoção de dois elementos.

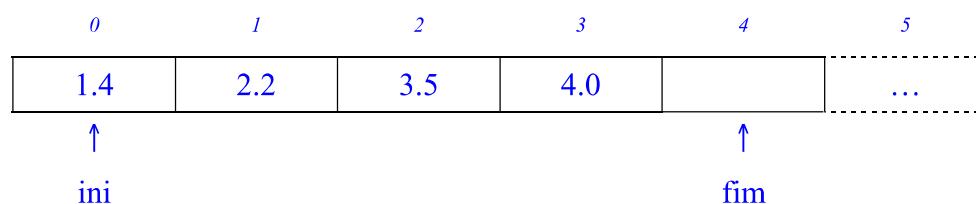


Figura 11.1: Fila após inserção de quatro novos elementos.

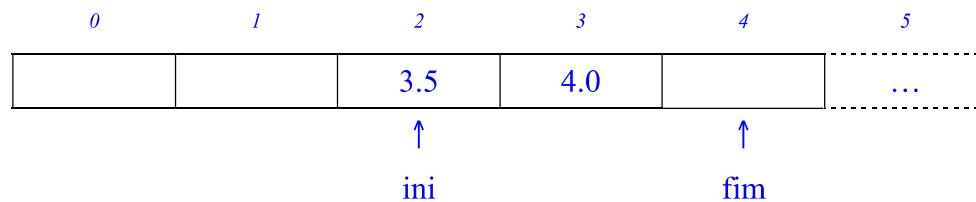


Figura 11.2: Fila após retirar dois elementos.

Com essa estratégia, é fácil observar que, em um dado instante, a parte ocupada do vetor pode chegar à última posição. Para reaproveitar as primeiras posições livres do vetor sem implementarmos uma re-arrumação trabalhosa dos elementos, podemos incrementar as posições do vetor de forma “circular”: se o último elemento da fila ocupa a última posição do vetor, inserimos os novos elementos a partir do início do vetor. Desta forma, em um dado momento, poderíamos ter quatro elementos, 20.0, 20.8, 21.2 e 24.3, distribuídos dois no fim do vetor e dois no início.

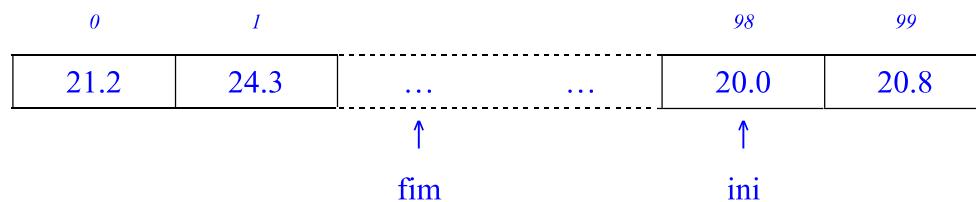


Figura 11.3: Fila com incremento circular.

Para essa implementação, os índices do vetor são incrementados de maneira que seus valores progridam “circularmente”. Desta forma, se temos 100 posições no vetor, os valores dos índices assumem os seguintes valores:

```
0, 1, 2, 3, ..., 98, 99, 0, 1, 2, 3, ..., 98, 99, 0, 1,...
```

Podemos definir uma função auxiliar responsável por incrementar o valor de um índice. Essa função recebe o valor do índice atual e fornece com valor de retorno o índice incrementado, usando o incremento circular. Uma possível implementação dessa função é:

```
int incr (int i)
{
    if (i == N-1)
        return 0;
    else
        return i+1;
}
```

Essa mesma função pode ser implementada de uma forma mais compacta, usando o operador módulo:

```
int incr(int i)
{
    return (i+1)%N;
}
```

Com o uso do operador módulo, muitas vezes optamos inclusive por dispensar a função auxiliar e escrever diretamente o incremento circular:

```
...
i=(i+1)%N;
...
```

Podemos declarar o tipo fila como sendo uma estrutura com três componentes: um vetor `vet` de tamanho `N`, um índice `ini` para o início da fila e um índice `fim` para o fim da fila.

Conforme ilustrado nas figuras acima, usamos as seguintes convenções para a identificação da fila:

`ini` marca a posição do próximo elemento a ser retirado da fila;
`fim` marca a posição (vazia), onde será inserido o próximo elemento.

Desta forma, a fila vazia se caracteriza por ter `ini == fim` e a fila cheia (quando não é possível inserir mais elementos) se caracteriza por ter `fim` e `ini` em posições consecutivas (circularmente): `incr(fim) == ini`. Note que, com essas convenções, a posição indicada por `fim` permanece sempre vazia, de forma que o número máximo de elementos na fila é `N-1`. Isto é necessário porque a inserção de mais um elemento faria `ini == fim`, e haveria uma ambigüidade entre fila cheia e fila vazia. Outra estratégia possível consiste em armazenar uma informação adicional, `n`, que indicaria explicitamente o número de elementos armazenados na fila. Assim, a fila estaria vazia se `n == 0` e cheia se `n == N-1`. Nos exemplos que se seguem, optamos por não armazenar `n` explicitamente.

A estrutura de fila pode então ser dada por:

```
#define N 100

struct fila {
    int ini, fim;
    float vet[N];
};
```

A função para criar a fila aloca dinamicamente essa estrutura e inicializa a fila como sendo vazia, isto é, com os índices `ini` e `fim` iguais entre si (no caso, usamos o valor zero).

```
Fila* cria (void)
{
    Fila* f = (Fila*) malloc(sizeof(Fila));
    f->ini = f->fim = 0; /* inicializa fila vazia */
    return f;
}
```

Para inserir um elemento na fila, usamos a próxima posição livre do vetor, indicada por `fim`. Devemos ainda assegurar que há espaço para a inserção do novo elemento, tendo em vista que trata-se de um vetor com capacidade limitada. Consideraremos que a função auxiliar que faz o incremento circular está disponível.

```
void insere (Fila* f, float v)
{
    if (incr(f->fim) == f->ini) { /* fila cheia: capacidade esgotada */
        printf("Capacidade da fila estourou.\n");
        exit(1); /* aborta programa */
    }
    /* insere elemento na próxima posição livre */
    f->vet[f->fim] = v;
    f->fim = incr(f->fim);
}
```

A função para retirar o elemento do início da fila fornece o valor do elemento retirado como retorno. Podemos também verificar se a fila está ou não vazia.

```
float retira (Fila* f)
{
    float v;
    if (vazia(f)) {
        printf("Fila vazia.\n");
        exit(1); /* aborta programa */
    }
    /* retira elemento do início */
    v = f->vet[f->ini];
    f->ini = incr(f->ini);
    return v;
}
```

A função que verifica se a fila está vazia pode ser dada por:

```
int vazia (Fila* f)
{
    return (f->ini == f->fim);
```

Finalmente, a função para liberar a memória alocada pela fila pode ser:

```
void libera (Fila* f)
{
    free(f);
}
```

12.3. Implementação de fila com lista

Vamos agora ver como implementar uma fila através de uma lista encadeada, que será, como nos exemplos anteriores, uma lista simplesmente encadeada, em que cada nó guarda um ponteiro para o próximo nó da lista. Como teremos que inserir e retirar elementos das extremidades opostas da lista, que representarão o início e o fim da fila, teremos que usar dois ponteiros, `ini` e `fim`, que apontam respectivamente para o primeiro e para o último elemento da fila. Essa situação é ilustrada na figura abaixo:

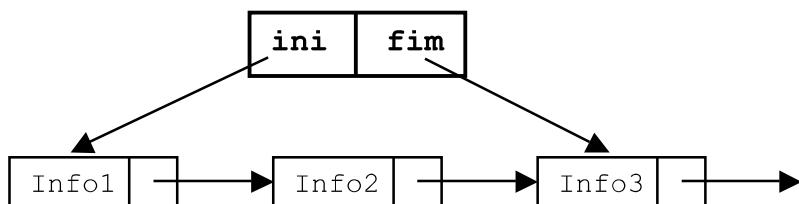


Figura 11.4: Estrutura de fila com lista encadeada.

A operação para retirar um elemento se dá no início da lista (fila) e consiste essencialmente em fazer com que, após a remoção, `ini` aponte para o sucessor do nó retirado. (Observe que seria mais complicado remover um nó do fim da lista, porque o antecessor de um nó não é encontrado com a mesma facilidade que seu sucessor.) A inserção também é simples, pois basta acrescentar à lista um sucessor para o último nó, apontado por `fim`, e fazer com que `fim` aponte para este novo nó.

O nó da lista para armazenar valores reais, como já vimos, pode ser dado por:

```
struct no {
    float info;
    struct no* prox;
};

typedef struct no No;
```

A estrutura da fila agrupa os ponteiros para o início e o fim da lista:

```
struct fila {
    No* ini;
    No* fim;
};
```

A função `cria` aloca a estrutura da fila e inicializa a lista como sendo vazia.

```
Fila* cria (void)
{
    Fila* f = (Fila*) malloc(sizeof(Fila));
    f->ini = f->fim = NULL;
    return f;
}
```

Cada novo elemento é inserido no fim da lista e, sempre que solicitado, retiramos o elemento do início da lista. Desta forma, precisamos de duas funções auxiliares de lista:

para inserir no fim e para remover do início. A função para inserir no fim ainda não foi discutida, mas é simples, uma vez que temos explicitamente armazenado o ponteiro para o último elemento. Essa função deve ter como valor de retorno o novo “fim” da lista. A função para retirar do início é idêntica à função usada na implementação de pilha.

```
/* função auxiliar: insere no fim */
No* ins_fim (No* fim, float v)
{
    No* p = (No*) malloc(sizeof(No));
    p->info = v;
    p->prox = NULL;
    if (fim != NULL) /* verifica se lista não estava vazia */
        fim->prox = p;
    return p;
}

/* função auxiliar: retira do início */
No* ret_ini (No* ini)
{
    No* p = ini->prox;
    free(ini);
    return p;
}
```

As funções que manipulam a fila fazem uso dessas funções de lista. Devemos salientar que a função de inserção deve atualizar ambos os ponteiros, `ini` e `fim`, quando da inserção do primeiro elemento. Analogamente, a função para retirar deve atualizar ambos se a fila tornar-se vazia após a remoção do elemento:

```
void insere (Fila* f, float v)
{
    f->fim = ins_fim(f->fim,v);
    if (f->ini==NULL) /* fila antes vazia? */
        f->ini = f->fim;
}

float retira (Fila* f)
{
    float v;
    if (vazia(f)) {
        printf("Fila vazia.\n");
        exit(1); /* aborta programa */
    }
    v = f->ini->info;
    f->ini = ret_ini(f->ini);
    if (f->ini == NULL) /* fila ficou vazia? */
        f->fim = NULL;
    return v;
}
```

A fila estará vazia se a lista estiver vazia:

```
int vazia (Fila* f)
{
    return (f->ini==NULL);
```

Por fim, a função que libera a fila deve antes liberar todos os elementos da lista.

```
void libera (Fila* f)
{
    No* q = f->ini;
    while (q!=NULL) {
        No* t = q->prox;
        free(q);
        q = t;
    }
    free(f);
}
```

Analogamente à pilha, para testar o código, pode ser útil implementarmos uma função que imprima os valores armazenados na fila. Os códigos abaixo ilustram a implementação dessa função nas duas versões de fila (vetor e lista). A ordem de impressão adotada é do início para o fim.

```
/* imprime: versão com vetor */
void imprime (Fila* f)
{
    int i;
    for (i=f->ini; i!=f->fim; i=incr(i))
        printf("%f\n", f->vet[i]);
}

/* imprime: versão com lista */
void imprime (Fila* f)
{
    No* q;
    for (q=f->ini; q!=NULL; q=q->prox)
        printf("%f\n", q->info);
}
```

Um exemplo simples de utilização da estrutura de fila é apresentado a seguir:

```
/* Módulo para ilustrar utilização da fila */

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

int main (void)
{
    Fila* f = cria();
    insere(f,20.0);
    insere(f,20.8);
    insere(f,21.2);
    insere(f,24.3);
    printf("Primeiro elemento: %f\n", retira(f));
    printf("Segundo elemento: %f\n", retira(f));
    printf("Configuracao da fila:\n");
    imprime(f);
    libera(f);
    return 0;
}
```

12.4. Fila dupla

A estrutura de dados que chamamos de *fila dupla* consiste numa fila na qual é possível inserir novos elementos em ambas as extremidades, no início e no fim. Conseqüentemente, permite-se também retirar elementos de ambos os extremos. É como se, dentro de uma mesma estrutura de fila, tivéssemos duas filas, com os elementos dispostos em ordem inversa uma da outra.

A interface do tipo abstrato que representa uma fila dupla acrescenta novas funções para inserir e retirar elementos. Podemos enumerar as seguintes operações:

- criar uma estrutura de fila dupla;
- inserir um elemento no início;
- inserir um elemento no fim;
- retirar o elemento do início;
- retirar o elemento do fim;
- verificar se a fila está vazia;
- liberar a fila.

O arquivo `fila2.h`, que representa a interface do tipo, pode conter o seguinte código:

```
typedef struct fila2 Fila2;

Fila2* cria (void);
void insere_ini (Fila2* f, float v);
void insere_fim (Fila2* f, float v);
float retira_ini (Fila2* f);
float retira_fim (Fila2* f);
int vazia (Fila2* f);
void libera (Fila2* f);
```

A implementação dessa estrutura usando um vetor para armazenar os elementos não traz grandes dificuldades, pois o vetor permite acesso randômico aos elementos, e fica como exercício.

Exercício: Implementar a estrutura de fila dupla com vetor.

Obs: Note que o decremento circular não pode ser feito de maneira compacta como fizemos para incrementar. Devemos decrementar o índice de uma unidade e testar se ficou negativo, atribuindo-lhe o valor $N-1$ em caso afirmativo.

12.5. Implementação de fila dupla com lista

A implementação de uma fila dupla com lista encadeada merece uma discussão mais detalhada. A dificuldade que encontramos reside na implementação da função para retirar um elemento do final da lista. Todas as outras funções já foram discutidas e poderiam ser facilmente implementadas usando uma lista simplesmente encadeada. No entanto, na lista simplesmente encadeada, a função para retirar do fim não pode ser implementada de forma eficiente, pois, dado o ponteiro para o último elemento da lista, não temos como acessar o anterior, que passaria a ser o então último elemento.

Para solucionar esse problema, temos que lançar mão da estrutura de lista duplamente encadeada (veja a seção 9.5). Nessa lista, cada nó guarda, além da referência para o próximo elemento, uma referência para o elemento anterior: dado o ponteiro de um nó, podemos acessar ambos os elementos adjacentes. Este arranjo resolve o problema de acessarmos o elemento anterior ao último. Devemos salientar que o uso de uma lista duplamente encadeada para implementar a fila é bastante simples, pois só manipulamos os elementos das extremidades da lista.

O arranjo de memória para implementarmos a fila dupla com lista é ilustrado na figura abaixo:

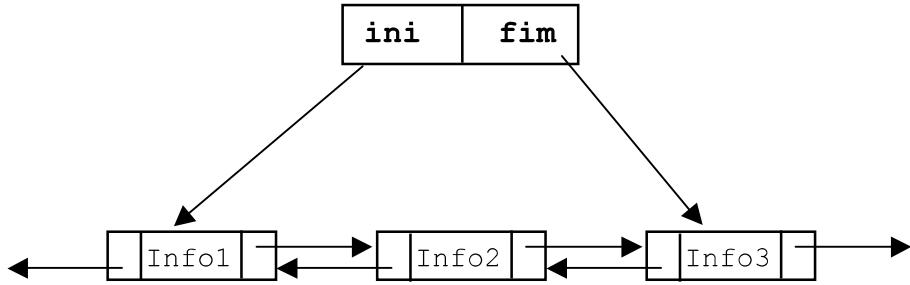


Figura 11.5: Arranjo da estrutura de fila dupla com lista.

O nó da lista duplamente encadeada para armazenar valores reais pode ser dado por:

```
struct no2 {
    float info;
    struct no2* ant;
    struct no2* prox;
};

typedef struct no2 No2;
```

A estrutura da fila dupla agrupa os ponteiros para o início e o fim da lista:

```
struct fila2 {
    No2* ini;
    No2* fim;
};
```

Interessa-nos discutir as funções para inserir e retirar elementos. As demais são praticamente idênticas às de fila simples. Podemos inserir um novo elemento em qualquer extremidade da fila. Portanto, precisamos de duas funções auxiliares de lista: para inserir no início e para inserir no fim. Ambas as funções são simples e já foram exaustivamente discutidas para o caso da lista simples. No caso da lista duplamente encadeada, a diferença consiste em termos que atualizar também o encadeamento para o elemento anterior. Uma possível implementação dessas funções é mostrada a seguir. Essas funções retornam, respectivamente, o novo nó inicial e final.

```
/* função auxiliar: insere no início */
No2* ins2_ini (No2* ini, float v) {
    No2* p = (No2*) malloc(sizeof(No2));
    p->info = v;
    p->prox = ini;
    p->ant = NULL;
    if (ini != NULL) /* verifica se lista não estava vazia */
        ini->ant = p;
    return p;
}

/* função auxiliar: insere no fim */
No2* ins2_fim (No2* fim, float v) {
    No2* p = (No2*) malloc(sizeof(No2));
    p->info = v;
    p->prox = NULL;
    p->ant = fim;
    if (fim != NULL) /* verifica se lista não estava vazia */
        fim->prox = p;
    return p;
}
```

Uma possível implementação das funções para remover o elemento do início ou do fim é mostrada a seguir. Essas funções também retornam, respectivamente, o novo nó inicial e final.

```
/* função auxiliar: retira do início */
No2* ret2_ini (No2* ini) {
    No2* p = ini->prox;
    if (p != NULL) /* verifica se lista não ficou vazia */
        p->ant = NULL;
    free(ini);
    return p;
}

/* função auxiliar: retira do fim */
No2* ret2_fim (No2* fim) {
    No2* p = fim->ant;
    if (p != NULL) /* verifica se lista não ficou vazia */
        p->prox = NULL;
    free(fim);
    return p;
}
```

As funções que manipulam a fila fazem uso dessas funções de lista, atualizando os ponteiros `ini` e `fim` quando necessário.

```
void insere_ini (Fila2* f, float v) {
    f->ini = ins2_ini(f->ini,v);
    if (f->fim==NULL) /* fila antes vazia? */
        f->fim = f->ini;
}

void insere_fim (Fila2* f, float v) {
    f->fim = ins2_fim(f->fim,v);
    if (f->ini==NULL) /* fila antes vazia? */
        f->ini = f->fim;
}

float retira_ini (Fila2* f) {
    float v;
    if (vazia(f)) {
        printf("Fila vazia.\n");
        exit(1); /* aborta programa */
    }
    v = f->ini->info;
    f->ini = ret2_ini(f->ini);
    if (f->ini == NULL) /* fila ficou vazia? */
        f->fim = NULL;
    return v;
}

float retira_fim (Fila2* f) {
    float v;
    if (vazia(f)) {
        printf("Fila vazia.\n");
        exit(1); /* aborta programa */
    }
    v = f->fim->info;
    f->fim = ret2_fim(f->fim);
    if (f->fim == NULL) /* fila ficou vazia? */
        f->ini = NULL;
    return v;
}
```

13. Árvores

W. Celes e J. L. Rangel

Nos capítulos anteriores examinamos as estruturas de dados que podem ser chamadas de unidimensionais ou lineares, como vetores e listas. A importância dessas estruturas é inegável, mas elas não são adequadas para representarmos dados que devem ser dispostos de maneira hierárquica. Por exemplo, os arquivos (documentos) que criamos num computador são armazenados dentro de uma estrutura hierárquica de diretórios (pastas). Existe um diretório base dentro do qual podemos armazenar diversos sub-diretórios e arquivos. Por sua vez, dentro dos sub-diretórios, podemos armazenar outros sub-diretórios e arquivos, e assim por diante, recursivamente. A Figura 13.1 mostra uma imagem de uma árvore de diretório no Windows 2000.

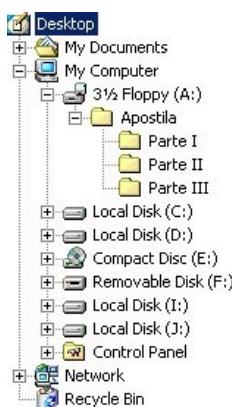


Figura 13.1: Um exemplo de árvore de diretório.

Neste capítulo, vamos introduzir *árvore*s, que são estruturas de dados adequadas para a representação de hierarquias. A forma mais natural para definirmos uma estrutura de árvore é usando recursividade. Uma árvore é composta por um conjunto de nós. Existe um nó r , denominado *raiz*, que contém zero ou mais sub-árvores, cujas raízes são ligadas diretamente a r . Esses nós raízes das sub-árvores são ditos *filhos* do nó *pai*, r . Nós com filhos são comumente chamados de *nós internos* e nós que não têm filhos são chamados de *folhas*, ou nós externos. É tradicional desenhar as árvores com a raiz para cima e folhas para baixo, ao contrário do que seria de se esperar. A Figura 13.2 exemplifica a estrutura de uma árvore.

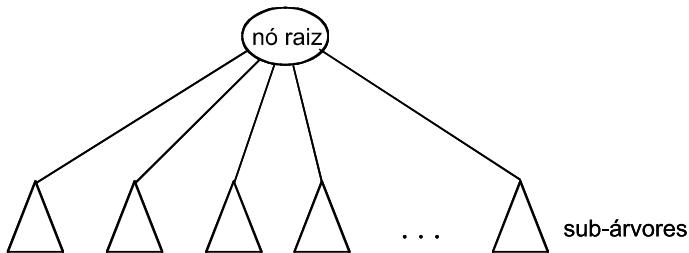


Figura 13.2: Estrutura de árvore.

Observamos que, por adotarmos essa forma de representação gráfica, não representamos explicitamente a direção dos ponteiros, subentendendo que eles apontam sempre do pai para os filhos.

O número de filhos permitido por nó e as informações armazenadas em cada nó diferenciam os diversos tipos de árvores existentes. Neste capítulo, estudaremos dois tipos de árvores. Primeiro, examinaremos as árvores binárias, onde cada nó tem, no máximo, dois filhos. Depois examinaremos as chamadas árvores genéricas, onde o número de filhos é indefinido. Estruturas recursivas serão usadas como base para o estudo e a implementação das operações com árvores.

13.1. Árvores binárias

Um exemplo de utilização de árvores binárias está na avaliação de expressões. Como trabalhamos com operadores que esperam um ou dois operandos, os nós da árvore para representar uma expressão têm no máximo dois filhos. Nessa árvore, os nós folhas representam operandos e os nós internos operadores. Uma árvore que representa, por exemplo a expressão $(3+6) * (4-1) + 5$ é ilustrada na Figura 13.3.

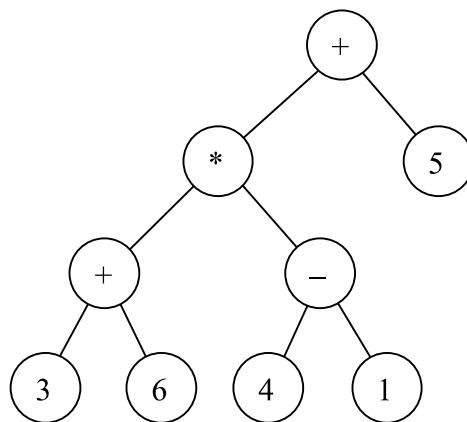


Figura 13.3: Árvore da expressão: $(3+6) * (4-1) + 5$.

Numa árvore binária, cada nó tem zero, um ou dois filhos. De maneira recursiva, podemos definir uma árvore binária como sendo:

- uma árvore vazia; ou
- um nó raiz tendo duas sub-árvores, identificadas como a sub-árvore da direita (*sad*) e a sub-árvore da esquerda (*sae*).

A Figura 13.4 ilustra a definição de árvore binária. Essa definição recursiva será usada na construção de algoritmos, e na verificação (informal) da correção e do desempenho dos mesmos.

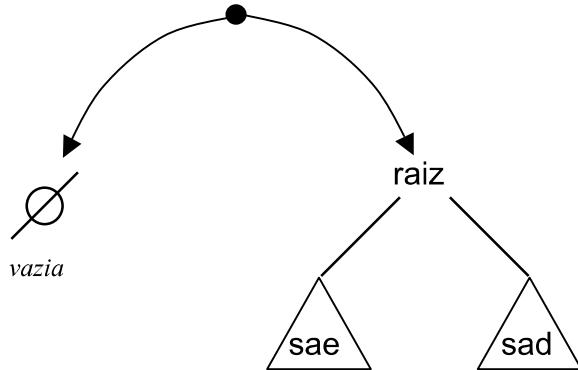


Figura 13.4: Representação esquemática da definição da estrutura de árvore binária.

A Figura 13.5 a seguir ilustra uma estrutura de árvore binária. Os nós a, b, c, d, e, f formam uma árvore binária da seguinte maneira: a árvore é composta do nó a , da sub-árvore à esquerda formada por b e d , e da sub-árvore à direita formada por c , e e f . O nó a representa a raiz da árvore e os nós b e c as raízes das sub-árvores. Finalmente, os nós d, e e f são folhas da árvore.

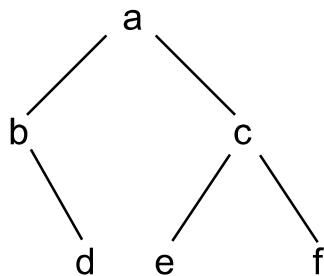


Figura 13.5: Exemplo de árvore binária

Para descrever árvores binárias, podemos usar a seguinte notação textual: a árvore vazia é representada por $<>$, e árvores não vazias por $<raiz\ sae\ sad>$. Com essa notação, a árvore da Figura 13.5 é representada por: $<a<d><>><c<e><>><f><>>>>$.

Pela definição, uma sub-árvore de uma árvore binária é sempre especificada como sendo a *sae* ou a *sad* de uma árvore maior, e qualquer das duas sub-árvores pode ser vazia. Assim, as duas árvores da Figura 13.6 são distintas.



Figura 13.6: Duas árvores binárias distintas.

Isto também pode ser visto pelas representações textuais das duas árvores, que são, respectivamente: $<a\ <>>\ <>\ >$ e $<a\ <>\ <>>\ >$.

Uma propriedade fundamental de todas as árvores é que só existe um caminho da raiz para qualquer nó. Com isto, podemos definir a *altura* de uma árvore como sendo o comprimento do caminho mais longo da raiz até uma das folhas. Por exemplo, a altura da árvore da Figura 13.5 é 2, e a altura das árvores da Figura 13.6 é 1. Assim, a altura de uma árvore com um único nó raiz é zero e, por conseguinte, dizemos que a altura de uma árvore vazia é negativa e vale -1.

Exercício: Mostrar que uma árvore binária de altura h tem, no mínimo, $h+1$ nós, e, no máximo, $2^{h+1} - 1$.

Representação em C

Análogo ao que fizemos para as demais estruturas de dados, podemos definir um tipo para representar uma árvore binária. Para simplificar a discussão, vamos considerar que a informação que queremos armazenar nos nós da árvore são valores de caracteres simples. Vamos inicialmente discutir como podemos representar uma estrutura de árvore binária em C. Que estrutura podemos usar para representar um nó da árvore? Cada nó deve armazenar três informações: a informação propriamente dita, no caso um caractere, e dois ponteiros para as sub-árvores, à esquerda e à direita. Então a estrutura de C para representar o nó da árvore pode ser dada por:

```
struct arv {
    char info;
    struct arv* esq;
    struct arv* dir;
};
```

Da mesma forma que uma lista encadeada é representada por um ponteiro para o primeiro nó, a estrutura da árvore como um todo é representada por um ponteiro para o nó raiz.

Como acontece com qualquer TAD (tipo abstrato de dados), as operações que fazem sentido para uma árvore binária dependem essencialmente da forma de utilização que se pretende fazer da árvore. Nesta seção, em vez de discutirmos a interface do tipo abstrato para depois mostrarmos sua implementação, vamos optar por discutir algumas operações mostrando simultaneamente suas implementações. Ao final da seção apresentaremos um arquivo que pode representar a interface do tipo. Nas funções que se seguem, consideraremos que existe o tipo `Arv` definido por:

```
typedef struct arv Arv;
```

Como veremos as funções que manipulam árvores são, em geral, implementadas de forma recursiva, usando a definição recursiva da estrutura.

Vamos procurar identificar e descrever apenas operações cuja utilidade seja a mais geral possível. Uma operação que provavelmente deverá ser incluída em todos os casos é a inicialização de uma árvore vazia. Como uma árvore é representada pelo endereço do nó raiz, uma árvore vazia tem que ser representada pelo valor `NULL`. Assim, a função que inicializa uma árvore vazia pode ser simplesmente:

```

Arv* inicializa(void)
{
    return NULL;
}

```

Para criar árvores não vazias, podemos ter uma operação que cria um nó raiz dadas a informação e suas duas sub-árvores, à esquerda e à direita. Essa função tem como valor de retorno o endereço do nó raiz criado e pode ser dada por:

```

Arv* cria(char c, Arv* sae, Arv* sad) {
    Arv* p=(Arv*)malloc(sizeof(Arv));
    p->info = c;
    p->esq = sae;
    p->dir = sad;
    return p;
}

```

As duas funções `inicializa` e `cria` representam os dois casos da definição recursiva de árvore binária: uma árvore binária (`Arv* a;`) é vazia (`a = inicializa();`) ou é composta por uma raiz e duas sub-árvores (`a = cria(c, sae, sad);`). Assim, com posse dessas duas funções, podemos criar árvores mais complexas.

Exemplo: Usando as operações `inicializa` e `cria`, crie uma estrutura que represente a árvore da Figura 13.5.

O exemplo da figura pode ser criada pela seguinte seqüência de atribuições.

```

Arv* a1= cria('d',inicializa(),inicializa()); /* sub-árvore com 'd'
*/
Arv* a2= cria('b',inicializa(),a1); /* sub-árvore com 'b'
*/
Arv* a3= cria('e',inicializa(),inicializa()); /* sub-árvore com 'e'
*/
Arv* a4= cria('f',inicializa(),inicializa()); /* sub-árvore com 'f'
*/
Arv* a5= cria('c',a3,a4); /* sub-árvore com 'c'
*/
Arv* a = cria('a',a2,a5); /* árvore com raiz 'a'
*/

```

Alternativamente, a árvore poderia ser criada com uma única atribuição, seguindo a sua estrutura, “recursivamente”:

```

Arv* a = cria('a',
               cria('b',
                     inicializa(),
                     cria('d', inicializa(), inicializa())
                   ),
               cria('c',
                     cria('e', inicializa(), inicializa()),
                     cria('f', inicializa(), inicializa())
                   )
               );

```

Para tratar a árvore vazia de forma diferente das outras, é importante ter uma operação que diz se uma árvore é ou não vazia. Podemos ter:

```

int vazia(Arv* a)
{
    return a==NULL;
}

```

Uma outra função muito útil consiste em exibir o conteúdo da árvore. Essa função deve percorrer recursivamente a árvore, visitando todos os nós e imprimindo sua informação. A implementação dessa função usa a definição recursiva da árvore. Vimos que uma árvore binária ou é vazia ou é composta pela raiz e por duas sub-árvore. Portanto, para imprimir a informação de todos os nós da árvore, devemos primeiro testar se a árvore é vazia. Se não for, imprimimos a informação associada a raiz e chamamos (recursivamente) a função para imprimir os nós das sub-árvore.

```

void imprime (Arv* a)
{
    if (!vazia(a)){
        printf("%c ", a->info);      /* mostra raiz */
        imprime(a->esq);           /* mostra sae */
        imprime(a->dir);          /* mostra sad */
    }
}

```

Exercício: (a) simule a chamada da função `imprime` aplicada à arvore ilustrada pela Figura 13.5 para verificar que o resultado da chamada é a impressão de a b d c e f. (b) Repita a experiência executando um programa que crie e mostre a árvore, usando o seu compilador de C favorito.

Exercício: Modifique a implementação de `imprime`, de forma que a saída impressa reflita, além do conteúdo de cada nó, a estrutura da árvore, usando a notação introduzida anteriormente. Assim, a saída da função seria:
`<a<d><>><c<e><>><f><>>>`.

Uma outra operação que pode ser acrescentada é a operação para liberar a memória alocada pela estrutura da árvore. Novamente, usaremos uma implementação recursiva. Um cuidado essencial a ser tomado é que as sub-árvore devem ser liberadas antes de se liberar o nó raiz, para que o acesso às sub-árvore não seja perdido antes de sua remoção. Neste caso, vamos optar por fazer com que a função tenha como valor de retorno a árvore atualizada, isto é, uma árvore vazia, representada por `NULL`.

```

Arv* libera (Arv* a) {
    if (!vazia(a)){
        libera(a->esq); /* libera sae */
        libera(a->dir); /* libera sad */
        free(a);          /* libera raiz */
    }
    return NULL;
}

```

Devemos notar que a definição de árvore, por ser recursiva, não faz distinção entre árvores e sub-árvore. Assim, `cria` pode ser usada para acrescentar (“enxertar”) uma sub-árvore em um ramo de uma árvore, e `libera` pode ser usada para remover (“podar”) uma sub-árvore qualquer de uma árvore dada.

Exemplo: Considerando a criação da árvore feita anteriormente:

```

Arv* a = cria('a',
               cria('b',
                     inicializa(),
                     cria('d', inicializa(), inicializa())
                   ),
               cria('c',
                     cria('e', inicializa(), inicializa()),
                     cria('f', inicializa(), inicializa())
                   )
                 );

```

Podemos acrescentar alguns nós, com:

```

a->esq->esq = cria('x',
                      cria('y', inicializa(), inicializa()),
                      cria('z', inicializa(), inicializa())
                    );

```

E podemos liberar alguns outros, com:

```
a->dir->esq = libera(a->dir->esq);
```

Deixamos como exercício a verificação do resultado final dessas operações.

É importante observar que, análogo ao que fizemos para a lista, o código cliente que chama a função libera é responsável por atribuir o valor atualizado retornado pela função, no caso uma árvore vazia. No exemplo acima, se não tivéssemos feito a atribuição, o endereço armazenado em r->dir->esq seria o de uma área de memória não mais em uso.

Exercício: Escreva uma função que percorre uma árvore binária para determinar sua altura. O protótipo da função pode ser dado por:

```
int altura(Arv* a);
```

Uma outra função que podemos considerar percorre a árvore buscando a ocorrência de um determinado caractere c em um de seus nós. Essa função tem como retorno um valor booleano (um ou zero) indicando a ocorrência ou não do caractere na árvore.

```

int busca (Arv* a, char c){
    if (vazia(a))
        return 0;      /* árvore vazia: não encontrou */
    else
        return a->info==c || busca(a->esq,c) || busca(a->dir,c);
}

```

Note que esta forma de programar busca, em C, usando o operador lógico `||` (“ou”) faz com que a busca seja interrompida assim que o elemento é encontrado. Isto acontece porque se `c==a->info` for verdadeiro, as duas outras expressões não chegam a ser avaliadas. Analogamente, se o caractere for encontrado na sub-árvore da esquerda, a busca não prossegue na sub-árvore da direita.

Podemos dizer que a expressão:

```
return c==a->info || busca(a->esq,c) || busca(a->dir,c);
```

é equivalente a:

```

if (c==a->info)
    return 1;
else if (busca(a->esq,c))
    return 1;
else
    return busca(a->dir,c);

```

Finalmente, considerando que as funções discutidas e implementadas acima formam a interface do tipo abstrato para representar uma árvore binária, um arquivo de interface arvbin.h pode ser dado por:

```

typedef struct Arv;

Arv* inicializa (void);
Arv* cria (char c, Arv* e, Arv* d);
int vazia (Arv* a);
void imprime (Arv* a);
Arv* libera (Arv* a);
int busca (Arv* a, char c);

```

Ordens de percurso em árvores binárias

A programação da operação `imprime`, vista anteriormente, seguiu a ordem empregada na definição de árvore binária para decidir a ordem em que as três ações seriam executadas:

Entretanto, dependendo da aplicação em vista, esta ordem poderia não ser a preferível, podendo ser utilizada uma ordem diferente desta, por exemplo:

```

imprime(a->esq);           /* mostra sae */
imprime(a->dir);           /* mostra sad */
printf("%c ", a->info);    /* mostra raiz */

```

Muitas operações em árvores binárias envolvem o percurso de todas as sub-árvores, executando alguma ação de tratamento em cada nó, de forma que é comum percorrer uma árvore em uma das seguintes ordens:

pré-ordem: trata *raiz*, percorre *sae*, percorre *sad*;

ordem simétrica: percorre *sae*, trata *raiz*, percorre *sad*;

pós-ordem: percorre *sae*, percorre *sad*, trata *raiz*.

Para função para liberar a árvore, por exemplo, tivemos que adotar a pós-ordem:

```

libera(a->esq); /* libera sae */
libera(a->dir); /* libera sad */
free(a);          /* libera raiz */

```

Na terceira parte do curso, quando tratarmos de árvores binárias de busca, apresentaremos um exemplo de aplicação de árvores binárias em que a ordem de percurso importante é a ordem simétrica. Algumas outras ordens de percurso podem ser definidas, mas a maioria das aplicações envolve uma dessas três ordens, percorrendo a *sae* antes da *sad*.

Exercício: Implemente versões diferentes da função `imprime`, percorrendo a árvore em ordem simétrica e em pós-ordem. Verifique o resultado da aplicação das duas funções na árvore da Figura 13.5.

13.2. Árvores genéricas

Nesta seção, vamos discutir as estruturas conhecidas como árvores genéricas. Como vimos, numa árvore binária o número de filhos dos nós é limitado em no máximo dois. No caso da árvore genérica, esta restrição não existe. Cada nó pode ter um número arbitrário de filhos. Essa estrutura deve ser usada, por exemplo, para representar uma árvore de diretórios.

Como veremos, as funções para manipularem uma árvore genérica também serão implementadas de forma recursiva, e serão baseadas na seguinte definição: uma árvore genérica é composta por:

- um nó raiz; e
- zero ou mais sub-árvores.

Estritamente, segundo essa definição, uma árvore não pode ser vazia, e a árvore vazia não é sequer mencionada na definição. Assim, uma folha de uma árvore não é um nó com sub-árvores vazias, como no caso da árvore binária, mas é um nó com *zero* sub-árvores. Em qualquer definição recursiva deve haver uma “condição de contorno”, que permita a definição de estruturas finitas, e, no nosso caso, a definição de uma árvore se encerra nas *folhas*, que são identificadas como sendo nós com zero sub-árvores.

Como as funções que implementaremos nesta seção serão baseadas nessa definição, não será considerado o caso de árvores vazias. Esta pequena restrição simplifica as implementações recursivas e, em geral, não limita a utilização da estrutura em aplicações reais. Uma árvore de diretório, por exemplo, nunca é vazia, pois sempre existe o diretório base – o diretório raiz.

Como as sub-árvores de um determinado nó formam um conjunto linear e são dispostas numa determinada ordem, faz sentido falarmos em primeira sub-árvore (sa_1), segunda sub-árvore (sa_2), etc. Um exemplo de uma árvore genérica é ilustrado na Figura 13.7.

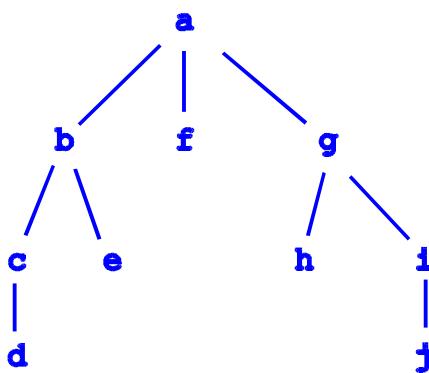


Figura 13.7: Exemplo de árvore genérica.

Nesse exemplo, podemos notar que o a árvore com raiz no nó a tem 3 sub-árvores, ou, equivalentemente, o nó a tem 3 filhos. Os nós b e g tem dois filhos cada um; os nós c e i tem um filho cada, e os nós d, e, h e j são folhas, e tem zero filhos.

De forma semelhante ao que foi feito no caso das árvores binárias, podemos representar essas árvores através de notação textual, seguindo o padrão: $\langle \text{raiz} \text{ } s_1 \text{ } s_2 \dots \text{ } s_n \rangle$. Com esta notação, a árvore da Figura 13.7 seria representada por:

$\alpha = \langle a \text{ } \langle b \text{ } \langle c \text{ } \langle d \rangle \rangle \text{ } \langle e \rangle \rangle \text{ } \langle f \rangle \text{ } \langle g \text{ } \langle h \rangle \rangle \text{ } \langle i \text{ } \langle j \rangle \rangle \rangle \rangle$

Podemos verificar que α representa a árvore do exemplo seguindo a seqüência de definição a partir das folhas:

```

 $\alpha_1 = \langle d \rangle$ 
 $\alpha_2 = \langle c \text{ } \alpha_1 \rangle = \langle c \text{ } \langle d \rangle \rangle$ 
 $\alpha_3 = \langle e \rangle$ 
 $\alpha_4 = \langle b \text{ } \alpha_2 \text{ } \alpha_3 \rangle = \langle b \text{ } \langle c \text{ } \langle d \rangle \rangle \text{ } \langle e \rangle \rangle$ 
 $\alpha_5 = \langle f \rangle$ 
 $\alpha_6 = \langle h \rangle$ 
 $\alpha_7 = \langle j \rangle$ 
 $\alpha_8 = \langle i \text{ } \alpha_7 \rangle = \langle i \text{ } \langle j \rangle \rangle$ 
 $\alpha_9 = \langle g \text{ } \alpha_6 \text{ } \alpha_8 \rangle = \langle g \text{ } \langle h \rangle \text{ } \langle i \text{ } \langle j \rangle \rangle \rangle$ 
 $\alpha = \langle a \text{ } \alpha_4 \text{ } \alpha_5 \text{ } \alpha_9 \rangle = \langle a \text{ } \langle b \text{ } \langle c \text{ } \langle d \rangle \rangle \text{ } \langle e \rangle \rangle \text{ } \langle f \rangle \text{ } \langle g \text{ } \langle h \rangle \rangle \text{ } \langle i \text{ } \langle j \rangle \rangle \rangle \rangle$ 

```

Representação em C

Dependendo da aplicação, podemos usar várias estruturas para representar árvores, levando em consideração o número de filhos que cada nó pode apresentar. Se soubermos, por exemplo, que numa aplicação o número máximo de filhos que um nó pode apresentar é 3, podemos montar uma estrutura com 3 campos para apontadores para os nós filhos, digamos, f_1 , f_2 e f_3 . Os campos não utilizados podem ser preenchidos com o valor nulo `NULL`, sendo sempre utilizados os campos em ordem. Assim, se o nó n tem 2 filhos, os campos f_1 e f_2 seriam utilizados, nessa ordem, para apontar para eles, ficando f_3 vazio. Prevendo um número máximo de filhos igual a 3, e considerando a implementação de árvores para armazenar valores de caracteres simples, a declaração do tipo que representa o nó da árvore poderia ser:

```

struct arv3 {
    char val;
    struct no *f1, *f2, *f3;
};

```

A Figura 13.8 indica a representação da árvore da Figura 13.7 com esta organização. Como se pode ver no exemplo, em cada um dos nós que tem menos de três filhos, o espaço correspondente aos filhos inexistentes é desperdiçado. Além disso, se não existe um limite superior no número de filhos, esta técnica pode não ser aplicável. O mesmo acontece se existe um limite no número de nós, mas esse limite será raramente alcançado, pois estariamos tendo um grande desperdício de espaço de memória com os campos não utilizados.

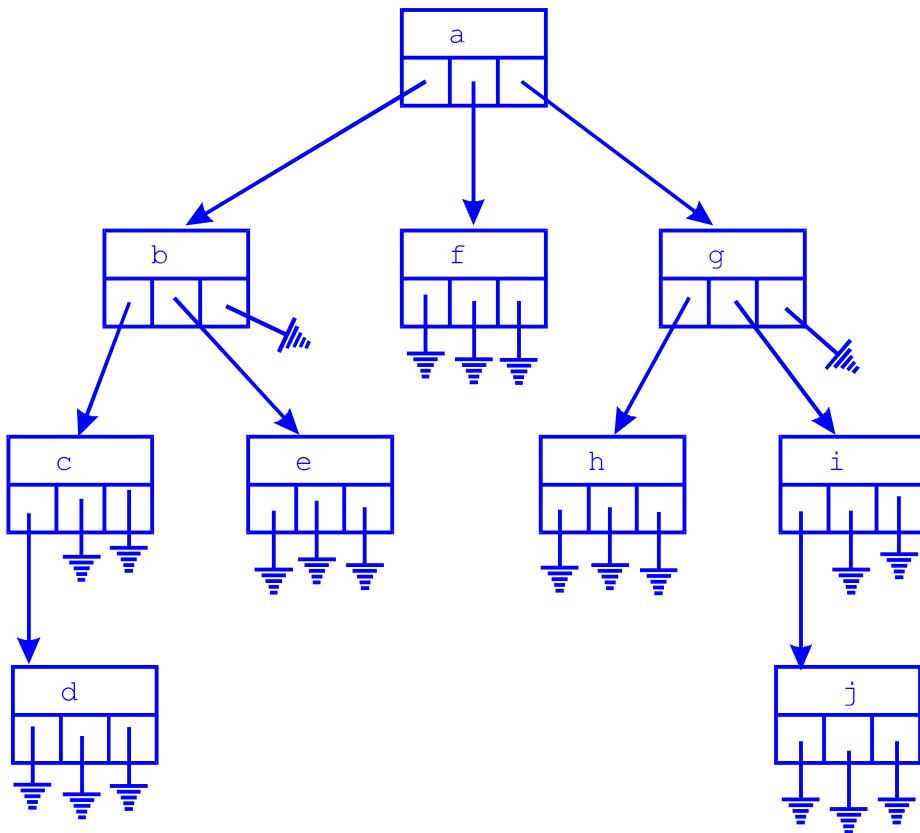


Figura 13.8: Árvore com no máximo três filhos por nó.

Uma solução que leva a um aproveitamento melhor do espaço utiliza uma “lista de filhos”: um nó aponta apenas para seu primeiro (`prim`) filho, e cada um de seus filhos, exceto o último, aponta para o próximo (`prox`) irmão. A declaração de um nó pode ser:

```
struct arvgen {
    char info;
    struct arvgen *prim;
    struct arvgen *prox;
};
```

A Figura 13.9 mostra o mesmo exemplo representado de acordo com esta estrutura. Uma das vantagens dessa representação é que podemos percorrer os filhos de um nó de forma sistemática, de maneira análoga ao que fizemos para percorrer os nós de uma lista simples.

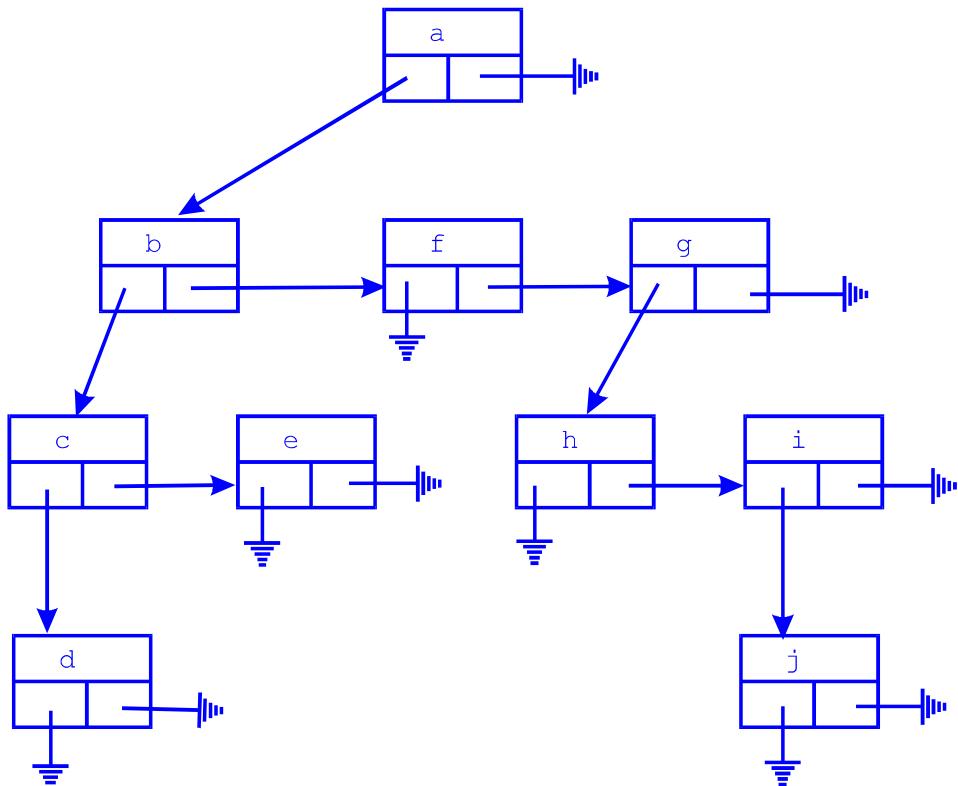


Figura 13.9: Exemplo usando “lista de filhos”.

Com o uso dessa representação, a generalização da árvore é apenas conceitual, pois, concretamente, a árvore foi transformada em uma árvore binária, com filhos esquerdos apontados por `prim` e direitos apontados por `prox`. Naturalmente, continuaremos a fazer referência aos nós nos termos da definição original. Por exemplo, os nós `b`, `f` e `g` continuarão a ser considerados *filhos* do nó `a`, como indicado na Figura 13.7, mesmo que a representação usada na Figura 13.9 os coloque a distâncias variáveis do nó pai.

Tipo abstrato de dado

Para exemplificar a implementação de funções que manipulam uma árvore genérica, vamos considerar a criação de um tipo abstrato de dados para representar árvores onde a informação associada a cada nó é um caractere simples. Nessa implementação, vamos optar por armazenar os filhos de um nó numa lista encadeada. Podemos definir o seguinte conjunto de operações:

- `cria`: cria um nó folha, dada a informação a ser armazenada;
- `insere`: insere uma nova sub-árvore como filha de um dado nó;
- `imprime`: percorre todos os nós e imprime suas informações;
- `busca`: verifica a ocorrência de um determinado valor num dos nós da árvore;
- `libera`: libera toda a memória alocada pela árvore.

A interface do tipo pode então ser definida no arquivo `arvgen.h` dado por:

```
typedef struct ArvGen ArvGen;

ArvGen* cria (char c);
void    insere (ArvGen* a, ArvGen* sa);
void    imprime (ArvGen* a);
int     busca (ArvGen* a, char c);
void    libera (ArvGen* a);
```

A estrutura `arvgen`, que representa o nó da árvore, é definida conforme mostrado anteriormente. A função para criar uma folha deve alocar o nó e inicializar seus campos, atribuindo `NULL` para os campos `prim` e `prox`, pois trata-se de um nó folha.

```
ArvGen* cria (char c)
{
    ArvGen *a = (ArvGen *) malloc(sizeof(ArvGen));
    a->info = c;
    a->prim = NULL;
    a->prox = NULL;
    return a;
}
```

A função que insere uma nova sub-árvore como filha de um dado nó é muito simples. Como não vamos atribuir nenhum significado especial para a posição de um nó filho, a operação de inserção pode inserir a sub-árvore em qualquer posição. Neste caso, vamos optar por inserir sempre no início da lista que, como já vimos, é a maneira mais simples de inserir um novo elemento numa lista encadeada.

```
void insere (ArvGen* a, ArvGen* sa)
{
    sa->prox = a->prim;
    a->prim = sa;
}
```

Com essas duas funções, podemos construir a árvore do exemplo da Figura 13.7 com o seguinte fragmento de código:

```
/* cria nós como folhas */
ArvGen* a = cria('a');
ArvGen* b = cria('b');
ArvGen* c = cria('c');
ArvGen* d = cria('d');
ArvGen* e = cria('e');
ArvGen* f = cria('f');
ArvGen* g = cria('g');
ArvGen* h = cria('h');
ArvGen* i = cria('i');
ArvGen* j = cria('j');
/* monta a hierarquia */
insere(c,d);
insere(b,e);
insere(b,c);
insere(i,j);
insere(g,i);
insere(g,h);
insere(a,g);
insere(a,f);
insere(a,b);
```

Para imprimir as informações associadas aos nós da árvore, temos duas opções para percorrer a árvore: pré-ordem, primeiro a raiz e depois as sub-árvore, ou pós-ordem, primeiro as sub-árvore e depois a raiz. Note que neste caso não faz sentido a ordem simétrica, uma vez que o número de sub-árvore é variável. Para essa função, vamos optar por imprimir o conteúdo dos nós em pré-ordem:

```
void imprime (ArvGen* a)
{
    ArvGen* p;
    printf("%c\n",a->info);
    for (p=a->prim; p!=NULL; p=p->prox)
        imprime(p);
}
```

A operação para buscar a ocorrência de uma dada informação na árvore é exemplificada abaixo:

```
int busca (ArvGen* a, char c)
{
    ArvGen* p;
    if (a->info==c)
        return 1;
    else {
        for (p=a->prim; p!=NULL; p=p->prox) {
            if (busca(p,c))
                return 1;
        }
    }
    return 0;
}
```

A última operação apresentada é a que libera a memória alocada pela árvore. O único cuidado que precisamos tomar na programação dessa função é a de liberar as subárvore antes de liberar o espaço associado a um nó (isto é, usar pós-ordem).

```
void libera (ArvGen* a)
{
    ArvGen* p = a->prim;
    while (p!=NULL) {
        ArvGen* t = p->prox;
        libera(p);
        p = t;
    }
    free(a);
}
```

Exercício: Escreva uma função com o protótipo

```
ArvGen* copia(ArvGen*a);
```

para criar dinamicamente uma cópia da árvore.

Exercício: Escreva uma função com o protótipo

```
int igual(ArvGen*a, ArvGen*b);
```

para testar se duas árvores são iguais.

14. Arquivos

W. Celes e J. L. Rangel

Neste capítulo, apresentaremos alguns conceitos básicos sobre arquivos, e alguns detalhes da forma de tratamento de arquivos em disco na linguagem C. A finalidade desta apresentação é discutir variadas formas para salvar (e recuperar) informações em arquivos. Com isto, será possível implementar funções para salvar (e recuperar) as informações armazenadas nas estruturas de dados que temos discutido.

Um arquivo em disco representa um elemento de informação do dispositivo de memória secundária. A memória secundária (disco) difere da memória principal em diversos aspectos. As duas diferenças mais relevantes são: eficiência e persistência. Enquanto o acesso a dados armazenados na memória principal é muito eficiente do ponto de vista de desempenho computacional, o acesso a informações armazenadas em disco é, em geral, extremamente ineficiente. Para contornar essa situação, os sistemas operacionais trabalham com *buffers*, que representam áreas da memória principal usadas como meio de transferência das informações de/para o disco. Normalmente, trechos maiores (alguns *kbytes*) são lidos e armazenados no *buffer* a cada acesso ao dispositivo. Desta forma, uma subsequente leitura de dados do arquivo, por exemplo, possivelmente não precisará acessar o disco, pois o dado requisitado pode já se encontrar no *buffer*. Os detalhes de como estes acessos se realizam dependem das características do dispositivo e do sistema operacional empregado.

A outra grande diferença entre memória principal e secundária (disco) consiste no fato das informações em disco serem persistentes, e em geral são lidas por programas e pessoas diferentes dos que as escreveram, o que faz com que seja mais prático atribuir nomes aos elementos de informação armazenados do disco (em vez de endereços), falando assim em arquivos e diretórios (pastas). Cada arquivo é identificado por seu nome e pelo diretório onde encontra-se armazenado numa determinada unidade de disco. Os nomes dos arquivos são, em geral, compostos pelo nome em si, seguido de uma extensão. A extensão pode ser usada para identificar a natureza da informação armazenada no arquivo ou para identificar o programa que gerou (e é capaz de interpretar) o arquivo. Assim, a extensão “.c” é usada para identificar arquivos que têm códigos fontes da linguagem C e a extensão “.doc” é, no Windows, usada para identificar arquivos gerados pelo editor Word da Microsoft.

Um arquivo pode ser visto de duas maneiras, na maioria dos sistemas operacionais: em “modo texto”, como um texto composto de uma seqüência de caracteres, ou em “modo binário”, como uma seqüência de bytes (números binários). Podemos optar por salvar (e recuperar) informações em disco usando um dos dois modos, texto ou binário. Uma vantagem do arquivo texto é que pode ser lido por uma pessoa e editado com editores de textos convencionais. Em contrapartida, com o uso de um arquivo binário é possível salvar (e recuperar) grandes quantidades de informação de forma bastante eficiente. O sistema operacional pode tratar arquivos “texto” de maneira diferente da utilizada para tratar arquivos “binários”. Em casos especiais, pode ser interessante tratar arquivos de um tipo como se fossem do outro, tomando os cuidados apropriados.

Para minimizar a dificuldade com que arquivos são manipulados, os sistemas operacionais oferecem um conjunto de serviços para ler e escrever informações do

disco. A linguagem C disponibiliza esses serviços para o programador através de um conjunto de funções. Os principais serviços que nos interessam são:

abertura de arquivos: o sistema operacional encontra o arquivo com o nome dado e prepara o buffer na memória.

leitura do arquivo: o sistema operacional recupera o trecho solicitado do arquivo. Como o buffer contém parte da informação do arquivo, parte ou toda a informação solicitada pode vir do buffer.

escrita no arquivo: o sistema operacional acrescenta ou altera o conteúdo do arquivo. A alteração no conteúdo do arquivo é feita inicialmente no buffer para depois ser transferida para o disco.

fechamento de arquivo: toda a informação constante do buffer é atualizada no disco e a área do buffer utilizada na memória é liberada.

Uma das informações que é mantida pelo sistema operacional é um ponteiro de arquivo (*file pointer*), que indica a posição de trabalho no arquivo. Para ler um arquivo, este apontador percorre o arquivo, do início até o fim, conforme os dados vão sendo recuperados (lidos) para a memória. Para escrever, normalmente, os dados são acrescentados quando o apontador se encontra no fim do arquivo.

Nas seções subsequentes, vamos apresentar as funções mais utilizadas em C para acessar arquivos e vamos discutir diferentes estratégias para tratar arquivos. Todas as funções da biblioteca padrão de C que manipulam arquivos encontram-se na biblioteca de entrada e saída, com interface em `stdio.h`.

14.1. Funções para abrir e fechar arquivos

A função básica para abrir um arquivo é `fopen`:

```
FILE* fopen (char* nome_arquivo, char* modo);
```

`FILE` é um tipo definido pela biblioteca padrão que representa uma abstração do arquivo. Quando abrimos um arquivo, a função tem como valor de retorno um ponteiro para o tipo `FILE`, e todas as operações subsequentes nesse arquivo receberão este endereço como parâmetro de entrada. Se o arquivo não puder ser aberto, a função tem como retorno o valor `NULL`.

Devemos passar o nome do arquivo a ser aberto. O nome do arquivo pode ser relativo, e o sistema procura o arquivo a partir do diretório corrente (diretório de trabalho do programa), ou pode ser absoluto, onde especificamos o nome completo do arquivo, incluindo os diretórios, desde o diretório raiz.

Existem diferentes modos de abertura de um arquivo. Podemos abrir um arquivo para leitura ou para escrita, e devemos especificar se o arquivo será aberto em modo texto ou em modo binário. O parâmetro `modo` da função `fopen` é uma cadeia de caracteres onde espera-se a ocorrência de caracteres que identificam o modo de abertura. Os caracteres interpretados no modo são:

r	<i>read-only</i>	Indica modo apenas para leitura, não pode ser alterado.
w	<i>write</i>	Indica modo para escrita.
a	<i>append</i>	Indica modo para escrita ao final do existente.
t	<i>text</i>	Indica modo texto.

b	<i>binary</i>	Indica modo binário.
---	---------------	----------------------

Se o arquivo já existe e solicitamos a sua abertura para escrita com modo `w`, o arquivo é apagado e um novo, inicialmente vazio, é criado. Quando solicitamos com modo `a`, o mesmo é preservado e novos conteúdos podem ser escritos no seu fim. Com ambos os modos, se o arquivo não existe, um novo é criado.

Os modos `b` e `t` podem ser combinados com os demais. Maiores detalhes e outros modos de abertura de arquivos podem ser encontrados nos manuais da linguagem C. Em geral, quando abrimos um arquivo, testamos o sucesso da abertura antes de qualquer outra operação, por exemplo:

```
...
FILE* fp;
fp = fopen("entrada.txt","rt");
if (fp == NULL) {
    printf("Erro na abertura do arquivo!\n");
    exit(1);
}
...
```

Após ler/escrever as informações de um arquivo, devemos fechá-lo. Para fechar um arquivo, devemos usar a função `fclose`, que espera como parâmetro o ponteiro do arquivo que se deseja fechar. O protótipo da função é:

```
int fclose (FILE* fp);
```

O valor de retorno dessa função é zero, se o arquivo for fechado com sucesso, ou a constante `EOF` (definida pela biblioteca), que indica a ocorrência de um erro.

14.2. Arquivos em modo texto

Nesta seção, vamos descrever as principais funções para manipular arquivos em modo texto. Também discutiremos algumas estratégias para organização de dados em arquivos.

Funções para ler dados

A principal função de C para leitura de dados em arquivos em modo texto é a função `fscanf`, similar à função `scanf` que temos usado para capturar valores entrados via o teclado. No caso da `fscanf`, os dados são capturados de um arquivo previamente aberto para leitura. A cada leitura, os dados correspondentes são transferidos para a memória e o ponteiro do arquivo avança, passando a apontar para o próximo dado do arquivo (que pode ser capturado numa leitura subsequente). O protótipo da função `fscanf` é:

```
int fscanf (FILE* fp, char* formato, ...);
```

Conforme pode ser observado, o primeiro parâmetro deve ser o ponteiro do arquivo do qual os dados serão lidos. Os demais parâmetros são os já discutidos para a função `scanf`: o formato e a lista de endereços de variáveis que armazenarão os valores lidos. Como a função `scanf`, a função `fscanf` também tem como valor de retorno o número de dados lidos com sucesso.

Uma outra função de leitura muito usada em modo texto é a função `fgetc` que, dado o ponteiro do arquivo, captura o próximo caractere do arquivo. O protótipo dessa função é:

```
int fgetc (FILE* fp);
```

Apesar do tipo do valor de retorno ser `int`, o valor retornado é o caractere lido. Se o fim do arquivo for alcançado, a constante `EOF` (*end of file*) é retornada.

Uma outra função muito utilizada para ler linhas de um arquivo é a função `fgets`. Essa função recebe como parâmetros três valores: a cadeia de caracteres que armazenará o conteúdo lido do arquivo, o número máximo de caracteres que deve ser lido e o ponteiro do arquivo. O protótipo da função é:

```
char* fgets (char* s, int n, FILE* fp);
```

A função lê do arquivo uma seqüência de caracteres, até que um caractere '`\n`' seja encontrado ou que o máximo de caracteres especificado seja alcançado. A especificação de um número máximo de caracteres é importante para evitarmos que se invada memória quando a linha do arquivo for maior do que supúnhamos. Assim, se dimensionarmos nossa cadeia de caracteres, que receberá o conteúdo da linha lida, com 121 caracteres, passaremos esse valor para a função, que lerá no máximo 120 caracteres, pois o último será ocupado pelo finalizador de *string* – o caractere '`\0`'. O valor de retorno dessa função é o ponteiro da própria cadeia de caracteres passada como parâmetro ou `NULL` no caso de ocorrer erro de leitura (por exemplo, quando alcançar o final do arquivo).

Funções para escrever dados

Dentre as funções que existem para escrever (salvar) dados em um arquivo, vamos considerar as duas mais freqüentemente utilizadas: `fprintf` e `fputc`, que são análogas, mas para escrita, às funções que vimos para leitura.

A função `fprintf` é análoga a função `printf` que temos usado para imprimir dados na saída padrão – em geral, o monitor. A diferença consiste na presença do parâmetro que indica o arquivo para o qual o dado será salvo. O valor de retorno dessa função representa o número de bytes escritos no arquivo. O protótipo da função é dado por:

```
int fprintf(FILE* fp, char* formato, ...);
```

A função `fputc` escreve um caractere no arquivo. O protótipo é:

```
int fputc (int c, FILE* fp);
```

O valor de retorno dessa função é o próprio caractere escrito, ou `EOF` se ocorrer um erro na escrita.

14.3. Estruturação de dados em arquivos textos

Existem diferentes formas para estruturarmos os dados em arquivos em modo texto, e diferentes formas de capturarmos as informações contidas neles. A forma de estruturar e

a forma de tratar as informações dependem da aplicação. A seguir, apresentaremos três formas de representarmos e acessarmos dados armazenados em arquivos: caractere a caractere, linha a linha, e usando palavras chaves.

Acesso caractere a caractere

Para exemplificar o acesso caractere a caractere, vamos discutir duas aplicações simples. Inicialmente, vamos considerar o desenvolvimento de um programa que conta as linhas de um determinado arquivo (para simplificar, vamos supor um arquivo fixo, com o nome “entrada.txt”). Para calcular o número de linhas do arquivo, podemos ler, caractere a caractere, todo o conteúdo do arquivo, contando o número de ocorrências do caractere que indica mudança de linha, isto é, o número de ocorrências do caractere '\n'.

```
/* Conta número de linhas de um arquivo */

#include <stdio.h>

int main (void)
{
    int c;
    int nlinhas = 0;      /* contador do número de linhas */
    FILE *fp;

    /* abre arquivo para leitura */
    fp = fopen("entrada.txt", "rt");
    if (fp==NULL) {
        printf("Não foi possível abrir arquivo.\n");
        return 1;
    }

    /* lê caractere a caractere */
    while ((c = fgetc(fp)) != EOF) {
        if (c == '\n')
            nlinhas++;
    }

    /* fecha arquivo */
    fclose(fp);

    /* exibe resultado na tela */
    printf("Número de linhas = %d\n", nlinhas);
}

return 0;
}
```

Como segundo exemplo, vamos considerar o desenvolvimento de um programa que lê o conteúdo do arquivo e cria um arquivo com o mesmo conteúdo, mas com todas as letras minúsculas convertidas para maiúsculas. Os nomes dos arquivos serão fornecidos, via teclado, pelo usuário. Uma possível implementação desse programa é mostrada a seguir:

```

/* Converte arquivo para maiúsculas */

#include <stdio.h>
#include <ctype.h> /* função toupper */

int main (void)
{
    int c;
    char entrada[121]; /* armazena nome do arquivo de entrada */
    char saída[121]; /* armazena nome do arquivo de saída */
    FILE* e; /* ponteiro do arquivo de entrada */
    FILE* s; /* ponteiro do arquivo de saída */

    /* pede ao usuário os nomes dos arquivos */
    printf("Digite o nome do arquivo de entrada: ");
    scanf("%120s", entrada);
    printf("Digite o nome do arquivo de saída: ");
    scanf("%120s", saída);

    /* abre arquivos para leitura e para escrita */
    e = fopen(entrada,"rt");
    if (e == NULL) {
        printf("Não foi possível abrir arquivo de entrada.\n");
        return 1;
    }
    s = fopen(saída,"wt");
    if (s == NULL) {
        printf("Não foi possível abrir arquivo de saída.\n");
        fclose(e);
        return 1;
    }

    /* lê da entrada e escreve na saída */
    while ((c = fgetc(e)) != EOF)
        fputc(toupper(c),s);

    /* fecha arquivos */
    fclose(e);
    fclose(s);

    return 0;
}

```

Acesso linha a linha

Em diversas aplicações, é mais adequado tratar o conteúdo do arquivo linha a linha. Um caso simples que podemos mostrar consiste em procurar a ocorrência de uma sub-cadeia de caracteres dentro de um arquivo (análogo a o que é feito pelo utilitário `grep` dos sistemas Unix). Se a sub-cadeia for encontrada, apresentamos como saída o número da linha da primeira ocorrência.

Para implementar esse programa, vamos utilizar a função `strstr`, que procura a ocorrência de uma sub-cadeia numa cadeia de caracteres maior. A função retorna o endereço da primeira ocorrência ou `NULL`, se a sub-cadeia não for encontrada. O protótipo dessa função é:

```
char* strstr (char* s, char* sub);
```

A nossa implementação consistirá em ler, linha a linha, o conteúdo do arquivo, contanto o número da linha. Para cada linha, verificamos se a ocorrência da sub-cadeia, interrompendo a leitura em caso afirmativo.

```

/* Procura ocorrência de sub-cadeia no arquivo */

#include <stdio.h>
#include <string.h> /* função strstr */

int main (void)
{
    int n = 0;          /* número da linha corrente */
    int achou = 0;      /* indica se achou sub-cadeia */
    char entrada[121]; /* armazena nome do arquivo de entrada */
    char subcadeia[121]; /* armazena sub-cadeia */
    char linha[121];   /* armazena cada linha do arquivo */
    FILE* fp;          /* ponteiro do arquivo de entrada */

    /* pede ao usuário o nome do arquivo e a sub-cadeia */
    printf("Digite o nome do arquivo de entrada: ");
    scanf("%120s", entrada);
    printf("Digite a sub-cadeia: ");
    scanf("%120s", subcadeia);

    /* abre arquivos para leitura */
    fp = fopen(entrada,"rt");
    if (fp == NULL) {
        printf("Não foi possível abrir arquivo de entrada.\n");
        return 1;
    }

    /* lê linha a linha */
    while (fgets(linha,121,fp) != NULL) {
        n++;
        if (strstr(linha,subcadeia) != NULL) {
            achou = 1;
            break;
        }
    }

    /* fecha arquivo */
    fclose(fp);

    /* exibe saída */
    if (achou)
        printf("Achou na linha %d.\n", n);
    else
        printf("Nao achou.");
}

return 0;
}

```

Como segundo exemplo de arquivos manipulados linha a linha, podemos citar o caso em que salvamos os dados com formatação por linha. Para exemplificar, vamos considerar que queremos salvar as informações da lista de figuras geométricas que discutimos na seção 9.3. A lista continha retângulos, triângulos e círculos.

Para salvar essas informações num arquivo, temos que escolher um formato apropriado, que nos permita posteriormente recuperar a informação salva. Para exemplificar um formato válido, vamos adotar uma formatação por linha: em cada linha salvamos um caractere que indica o tipo da figura (r, t ou c), seguido dos parâmetros que definem a figura, base e altura para os retângulos e triângulos ou raio para os círculos. Para

enriquecer o formato, podemos considerar que as linhas iniciadas com o caractere `#` representam comentários e devem ser desconsideradas na leitura. Por fim, linhas em branco são permitidas e desprezadas. Um exemplo do conteúdo de um arquivo com esse formato é apresentado na Figura 14.1 (note a presença de linhas em branco e linhas que são comentários):

```
# Lista de figuras geometricas

r 2.0 1.2
c 5.8
# t 1.23 12
t 4 1.02

c 5.1
```

Figura 14.1: Exemplo de formatação por linha.

Para recuperarmos as informações contidas num arquivo com esse formato, podemos ler do arquivo cada uma das linhas e depois ler os dados contidos na linha. Para tanto, precisamos introduzir uma função adicional muito útil. Trata-se da função que permite ler dados de uma cadeia de caracteres. A função `sscanf` é similar às funções `scanf` e `fscanf`, mas captura os valores armazenados numa *string*. O protótipo dessa função é:

```
int sscanf (char* s, char* formato, ...);
```

A primeira cadeia de caracteres passada como parâmetro representa a *string* da qual os dados serão lidos. Com essa função, é possível ler uma linha de um arquivo e depois ler as informações contidas na linha. (Analogamente, existe a função `sprintf` que permite escrever dados formatados numa *string*.)

Faremos a interpretação do arquivo da seguinte forma: para cada linha lida do arquivo, tentaremos ler do conteúdo da linha um caractere (desprezando eventuais caracteres brancos iniciais) seguido de dois números reais. Se nenhum dado for lido com sucesso, significa que temos uma linha vazia e devemos desprezá-la. Se pelo menos um dado (no caso, o caractere) for lido com sucesso, podemos interpretar o tipo da figura geométrica armazenada na linha, ou detectar a ocorrência de um comentário. Se for um retângulo ou um triângulo, os dois valores reais também deverão ter sido lidos com sucesso. Se for um círculo, apenas um valor real deverá ter sido lido com sucesso. O fragmento de código abaixo ilustra essa implementação. Supõe-se que `fp` representa um ponteiro para um arquivo com formato válido aberto para leitura, em modo texto.

```
char c;
float v1, v2;
FILE* fp;
char linha[121];
...
while (fgets(linha, 121, fp)) {
    int n = sscanf(linha, " %c %f %f", &c, &v1, &v2);
    if (n>0) {
        switch(c) {
            case '#':
                /* desprezar linha de comentário */
                break;
            case 'r':
                if (n!=3) {
                    /* tratar erro de formato do arquivo */
                }
        }
    }
}
```

```

        ...
    }
    else {
        /* interpretar retângulo: base = v1, altura = v2 */
        ...
    }
break;
case 't':
if (n!=3) {
    /* tratar erro de formato do arquivo */
    ...
}
else {
    /* interpretar triângulo: base = v1, altura = v2 */
    ...
}
break;
case 'c':
if (n!=2) {
    /* tratar erro de formato do arquivo */
    ...
}
else {
    /* interpretar círculo: raio = v1 */
    ...
}
break;
default:
    /* tratar erro de formato do arquivo */
    ...
break;
}
}
...

```

A rigor, para o formato descrito, não precisávamos fazer a interpretação do arquivo linha a linha. O arquivo poderia ter sido interpretado capturando-se inicialmente um caractere que então indicaria qual a próxima informação a ser lida. No entanto, em algumas situações a interpretação linha a linha ilustrada acima é a única forma possível. Para exemplificar, vamos considerar um arquivo que representa um conjunto de pontos no espaço 3D. Esses pontos podem ser dados pelas suas três coordenadas x , y e z . Um formato bastante flexível para esse arquivo considera que cada ponto é dado em uma linha e permite a omissão da terceira coordenada, se essa for igual a zero. Dessa forma, o formato atende também a descrição de pontos no espaço 2D. Um exemplo desse formato é ilustrado abaixo:

```

2.3  4.5  6.0
1.2  10.4
7.4  1.3  9.6
...

```

Para interpretar esse formato, devemos ler cada uma das linhas e tentar ler três valores reais de cada linha (aceitando o caso de apenas dois valores serem lidos com sucesso).

Exercício: Faça um programa que interprete o formato de pontos 3D descrito acima, armazenando-os num vetor.

Acesso via palavras chaves

Quando os objetos num arquivo têm descrições de tamanhos variados, é comum adotarmos uma formatação com o uso de palavras chaves. Cada objeto é precedido por uma palavra chave que o identifica. A interpretação desse tipo de arquivo pode ser feita lendo-se as palavras chaves e interpretando a descrição do objeto correspondente. Para ilustrar, vamos considerar que, além de retângulos, triângulos e círculos, também temos polígonos quaisquer no nosso conjunto de figuras geométricas. Cada polígono pode ser descrito pelo número de vértices que o compõe, seguido das respectivas coordenadas desses vértices. A Figura 14.2 ilustra esse formato.

```
RETANGULO
    b    h

TRIANGULO
    b    h

CIRCULO
    r

POLIGONO
    n
    x1  y1
    x2  y2
    ...
    xn  yn
```

Figura 14.2: Formato com uso de palavras chaves.

O fragmento de código a seguir ilustra a interpretação desse formato, onde `fp` representa o ponteiro para o arquivo aberto para leitura.

```
...
FILE* fp;
char palavra[121];
...
while (fscanf(fp,"%120s",palavra) == 1)
{
    if (strcmp(palavra,"RETANGULO")==0) {
        /* interpreta retângulo */
    }
    else if (strcmp(palavra,"TRIANGULO")==0) {
        /* interpreta triângulo */
    }
    else if (strcmp(palavra,"CIRCULO")==0) {
        /* interpreta círculo */
    }
    else if (strcmp(palavra,"POLIGONO")==0) {
        /* interpreta polígono */
    }
    else {
        /* trata erro de formato */
    }
}
```

14.4. Arquivos em modo binário

Arquivos em modo binário servem para salvarmos (e depois recuperarmos) o conteúdo da memória principal diretamente no disco. A memória é escrita copiando-se o conteúdo de cada byte da memória para o arquivo. Uma das grandes vantagens de se usar arquivos binários é que podemos salvar (e recuperar) uma grande quantidade de dados de forma bastante eficiente. Neste curso, vamos apenas apresentar as duas funções básicas para manipulação de arquivos binários.

Função para salvar e recuperar

Para escrever (salvar) dados em arquivos binários, usamos a função `fwrite`. O protótipo dessa função pode ser simplificado por¹:

```
int fwrite (void* p, int tam, int nelem, FILE* fp);
```

O primeiro parâmetro dessa função representa o endereço de memória cujo conteúdo deseja-se salvar em arquivo. O parâmetro `tam` indica o tamanho, em bytes, de cada elemento e o parâmetro `nelem` indica o número de elementos. Por fim, passa-se o ponteiro do arquivo binário para o qual o conteúdo da memória será copiado.

A função para ler (recuperar) dados de arquivos binários é análoga, sendo que agora o conteúdo do disco é copiado para o endereço de memória passado como parâmetro. O protótipo da função pode ser dado por:

```
int fread (void* p, int tam, int nelem, FILE* fp);
```

Para exemplificar a utilização dessas funções, vamos considerar que uma aplicação tem um conjunto de pontos armazenados num vetor. O tipo que define o ponto pode ser:

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

Uma função para salvar o conteúdo de um vetor de pontos pode receber como parâmetros o nome do arquivo, o número de pontos no vetor, e o ponteiro para o vetor. Uma possível implementação dessa função é ilustrada abaixo:

```
void salva (char* arquivo, int n, Ponto* vet)  
{  
    FILE* fp = fopen(arquivo,"wb");  
    if (fp==NULL) {  
        printf("Erro na abertura do arquivo.\n");  
        exit(1);  
    }  
    fwrite(vet,sizeof(Ponto),n,fp);  
    fclose(fp);  
}
```

¹ A rigor, os tipos `int` são substituídos pelo tipo `size_t`, definido pela biblioteca padrão, sendo, em geral, sinônimo para um inteiro sem sinal (`unsigned int`).

A função para recuperar os dados salvos pode ser:

```
void carrega (char* arquivo, int n, Ponto* vet)
{
    FILE* fp = fopen(arquivo,"rb");
    if (fp==NULL) {
        printf("Erro na abertura do arquivo.\n");
        exit(1);
    }
    fread(vet,sizeof(Ponto),n,fp);
    fclose(fp);
}
```

15. Ordenação

W. Celes e J. L. Rangel

Em diversas aplicações, os dados devem ser armazenados obedecendo uma determinada ordem. Alguns algoritmos podem explorar a ordenação dos dados para operar de maneira mais eficiente, do ponto de vista de desempenho computacional. Para obtermos os dados ordenados, temos basicamente duas alternativas: ou inserimos os elementos na estrutura de dados respeitando a ordenação (dizemos que a ordenação é garantida por construção), ou, a partir de um conjunto de dados já criado, aplicamos um algoritmo para ordenar seus elementos. Neste capítulo, vamos discutir dois algoritmos de ordenação que podem ser empregados em aplicações computacionais.

Devido ao seu uso muito freqüente, é importante ter à disposição algoritmos de ordenação (*sorting*) eficientes tanto em termos de tempo (devem ser rápidos) como em termos de espaço (devem ocupar pouca memória durante a execução). Vamos descrever os algoritmos de ordenação considerando o seguinte cenário:

- a entrada é um vetor cujos elementos precisam ser ordenados;
- a saída é o mesmo vetor com seus elementos na ordem especificada;
- o espaço que pode ser utilizado é apenas o espaço do próprio vetor.

Portanto, vamos discutir *ordenação de vetores*. Como veremos, os algoritmos de ordenação podem ser aplicados a qualquer informação, desde que exista uma ordem definida entre os elementos. Podemos, por exemplo, ordenar um vetor de valores inteiros, adotando uma ordem crescente ou decrescente. Podemos também aplicar algoritmos de ordenação em vetores que guardam informações mais complexas, por exemplo um vetor que guarda os dados relativos a alunos de uma turma, com nome, número de matrícula, etc. Nesse caso, a ordem entre os elementos tem que ser definida usando uma das informações do aluno como chave da ordenação: alunos ordenados pelo nome, alunos ordenados pelo número de matrícula, etc.

Nos casos em que a informação é complexa, raramente se encontra toda a informação relevante sobre os elementos do vetor no próprio vetor; em vez disso, cada componente do vetor pode conter apenas um ponteiro para a informação propriamente dita, que pode ficar em outra posição na memória. Assim, a ordenação pode ser feita sem necessidade de mover grandes quantidades de informação, para re-arrumar as componentes do vetor na sua ordem correta. Para trocar a ordem entre dois elementos, apenas os ponteiros são trocados. Em muitos casos, devido ao grande volume, a informação pode ficar em um arquivo de disco, e o elemento do vetor ser apenas uma referência para a posição da informação nesse arquivo.

Neste capítulo, examinaremos os algoritmos de ordenação conhecidos como “ordenação bolha” (*bubble sort*) e “ordenação rápida” (*quick sort*), ou, mais precisamente, versões simplificadas desses algoritmos.

15.1. Ordenação bolha

O algoritmo de “ordenação bolha”, ou “*bubble sort*”, recebeu este nome pela imagem pitoresca usada para descrevê-lo: os elementos maiores são mais leves, e sobem como bolhas até suas posições corretas. A idéia fundamental é fazer uma série de comparações entre os elementos do vetor. Quando dois elementos estão fora de ordem,

há uma inversão e esses dois elementos são trocados de posição, ficando em ordem correta. Assim, o primeiro elemento é comparado com o segundo. Se uma inversão for encontrada, a troca é feita. Em seguida, independente se houve ou não troca após a primeira comparação, o segundo elemento é comparado com o terceiro, e, caso uma inversão seja encontrada, a troca é feita. O processo continua até que o penúltimo elemento seja comparado com o último. Com este processo, garante-se que o elemento de maior valor do vetor será levado para a última posição. A ordenação continua, posicionando o segundo maior elemento, o terceiro, etc., até que todo o vetor esteja ordenado.

Para exemplificar, vamos considerar que os elementos do vetor que queremos ordenar são valores inteiros. Assim, consideremos a ordenação do seguinte vetor:

25 48 37 12 57 86 33 92

Seguimos os passos indicados:

25 48 37 12 57 86 33 92	25x48
25 48 37 12 57 86 33 92	48x37 troca
25 37 48 12 57 86 33 92	48x12 troca
25 37 12 48 57 86 33 92	48x57
25 37 12 48 57 86 33 92	57x86
25 37 12 48 57 86 33 92	86x33 troca
25 37 12 48 57 33 86 92	86x92
25 37 12 48 57 33 86 <u>92</u>	final da primeira passada

Neste ponto, o maior elemento, 92, já está na sua posição final.

25 37 12 48 57 33 86 <u>92</u>	25x37
25 37 12 48 57 33 86 <u>92</u>	37x12 troca
25 12 37 48 57 33 86 <u>92</u>	37x48
25 12 37 48 57 33 86 <u>92</u>	48x57
25 12 37 48 57 33 86 <u>92</u>	57x33 troca
25 12 37 48 33 57 86 <u>92</u>	57x86
25 12 37 48 33 57 86 <u>92</u>	final da segunda passada

Neste ponto, o segundo maior elemento, 86, já está na sua posição final.

25 12 37 48 33 57 <u>86</u> 92	25x12 troca
12 25 37 48 33 57 <u>86</u> 92	25x37
12 25 37 48 33 57 <u>86</u> 92	37x48
12 25 37 48 33 57 <u>86</u> 92	48x33 troca
12 25 37 33 48 57 <u>86</u> 92	48x57
12 25 37 33 48 57 <u>86</u> 92	final da terceira passada

Idem para 57.

12 25 37 33 48 <u>57</u> 86 92	12x25
12 25 37 33 48 <u>57</u> 86 92	25x37
12 25 37 33 48 <u>57</u> 86 92	37x33 troca
12 25 33 37 48 <u>57</u> 86 92	37x48
12 25 33 37 48 <u>57</u> 86 92	final da quarta passada

Idem para 48.

12 25 33 37 <u>48</u> 57 86 92	12x25
12 25 33 37 <u>48</u> 57 86 92	25x33
12 25 33 37 <u>48</u> 57 86 92	33x37
12 25 33 37 <u>48</u> 57 86 92	final da quinta passada

Idem para 37.

12 25 33 <u>37</u> 48 57 86 92	12x25
12 25 33 <u>37</u> 48 57 86 92	25x33
12 25 <u>33</u> <u>37</u> 48 57 86 92	final da sexta passada

Idem para 33.

12 25 33 37 48 57 86 92	12x25
<u>12</u> <u>25</u> 33 37 48 57 86 92	final da sétima passada

Idem para 25 e, consequentemente, 12.

12 25 33 37 48 57 86 92	final da ordenação
-------------------------	--------------------

A parte sabidamente já ordenada do vetor está sublinhada. Na realidade, após a troca de 37×33 , o vetor se encontra totalmente ordenado, mas esse fato não é levado em consideração por esta versão do algoritmo.

Uma função que implementa esse algoritmo é apresentada a seguir. A função recebe como parâmetros o número de elementos e o ponteiro do primeiro elemento do vetor que se deseja ordenar. Vamos considerar o ordenação de um vetor de valores inteiros.

```
/* Ordenação bolha */
void bolha (int n, int* v)
{
    int i,j;
    for (i=n-1; i>=1; i--)
        for (j=0; j<i; j++)
            if (v[j]>v[j+1]) { /* troca */
                int temp = v[j];
                v[j] = v[j+1];
                v[j+1] = temp;
            }
}
```

Uma função cliente para testar esse algoritmo pode ser dada por:

```
/* Testa algoritmo de ordenação bolha */
#include <stdio.h>

int main (void)
{
    int i;
    int v[8] = {25,48,37,12,57,86,33,92};
    bolha(8,v);
    printf("Vetor ordenado: ");
    for (i=0; i<8; i++)
        printf("%d ",v[i]);
    printf("\n");
    return 0;
}
```

Para evitar que o processo continue mesmo depois de o vetor estar ordenado, podemos interromper o processo quando houver uma passagem inteira sem trocas, usando uma variante do algoritmo apresentado acima:

```

/* Ordenação bolha (2a. versão) */
void bolha2 (int n, int* v)
{
    int i, j;
    for (i=n-1; i>0; i--) {
        int troca = 0;
        for (j=0; j<i; j++)
            if (v[j]>v[j+1]) { /* troca */
                int temp = v[j];
                v[j] = v[j+1];
                v[j+1] = temp;
                troca = 1;
            }
        if (troca == 0) /* nao houve troca */
            return;
    }
}

```

A variável `troca` guarda o valor 0 (falso) quando uma passada do vetor (no `for` interno) se faz sem nenhuma troca.

O esforço computacional despendido pela ordenação de um vetor por este procedimento é fácil de se determinar, pelo número de comparações, que serve também para estimar o número máximo de trocas que podem ser realizadas. Na primeira passada, fazemos $n-1$ comparações; na segunda, $n-2$; na terceira $n-3$; e assim por diante. Logo, o tempo total gasto pelo algoritmo é proporcional a $(n-1) + (n-2) + \dots + 2 + 1$. A soma desses termos é proporcional ao quadrado de n . Dizemos que o algoritmo é de ordem quadrática e representamos isso escrevendo $O(n^2)$.

Implementação recursiva

Analisando a forma com que a ordenação bolha funciona, verificamos que o algoritmo procura resolver o problema da ordenação por partes. Inicialmente, o algoritmo coloca em sua posição (no final do vetor) o maior elemento, e o problema restante é semelhante ao inicial, só que com um vetor com menos elementos, formado pelos elementos $v[0], \dots, v[n-2]$.

Baseado nessa observação, é fácil implementar um algoritmo de ordenação bolha recursivamente. Embora não seja a forma mais adequada de implementarmos esse algoritmo, o estudo dessa recursão nos ajudará a entender a idéia por trás do próximo algoritmo de ordenação que veremos mais adiante.

O algoritmo recursivo de ordenação bolha posiciona o elemento de maior valor e chama, recursivamente, o algoritmo para ordenar o vetor restante, com $n-1$ elementos.

```

/* Ordenação bolha recursiva */
void bolha_rec (int n, int* v)
{
    int j;
    int troca = 0;
    for (j=0; j<n-1; j++)
        if (v[j]>v[j+1]) { /* troca */
            int temp = v[j];
            v[j] = v[j+1];
            v[j+1] = temp;
            troca = 1;
        }
    if (troca != 0) /* houve troca */
        bolha_rec(n-1,v);
}

```

Algoritmo genérico**

Esse mesmo algoritmo pode ser aplicado a vetores que guardam outras informações. O código escrito acima pode ser reaproveitado, a menos de alguns detalhes. Primeiro, a assinatura da função deve ser alterada, pois deixamos de ter um vetor de inteiros; segundo, a forma de comparação entre os elementos também deve ser alterada, pois não podemos, por exemplo, comparar duas cadeias de caracteres usando simplesmente o operador relacional “maior que” (>).

Para aumentar o potencial de reuso do nosso código, podemos re-escrever o algoritmo de ordenação apresentado acima tornando-o independente da informação armazenada no vetor. Vamos inicialmente discutir como podemos abstrair a função de comparação. O mesmo algoritmo para ordenação de inteiros apresentado acima pode ser re-escrito usando-se uma função auxiliar que faz a comparação. Em vez de compararmos diretamente dois elementos com o operador “maior que”, usamos uma função auxiliar que, dados dois elementos, verifica se o primeiro é maior que o segundo.

```

/* Função auxiliar de comparação */
int compara (int a, int b)
{
    if (a > b)
        return 1;
    else
        return 0;
}
/* Ordenação bolha (3a. versão) */
void bolha (int n, int* v)
{
    int i, j;
    for (i=n-1; i>0; i--) {
        int troca = 0;
        for (j=0; j<i; j++)
            if (compara(v[j],v[j+1])) { /* troca */
                int temp = v[j];
                v[j] = v[j+1];
                v[j+1] = temp;
                troca = 1;
            }
        if (troca == 0) /* nao houve troca */
            return;
    }
}

```

Desta forma, já aumentamos o potencial de reuso do algoritmo. Podemos, por exemplo, arrumar os elementos em ordem decrescente simplesmente re-escrevendo a função compara. A idéia fundamental é escrever uma função de comparação que recebe dois

elementos e verifica se há uma inversão de ordem entre o primeiro e o segundo. Assim, se tivéssemos um vetor de cadeia de caracteres para ordenar, poderíamos usar a seguinte função de ordenação.

```
int compara (char* a, char* b)
{
    if (strcmp(a,b) > 0)
        return 1;
    else
        return 0;
}
```

Consideremos agora um vetor de ponteiros para a estrutura Aluno:

```
struct aluno {
    char nome[81];
    char mat[8];
    char turma;
    char email[41];
};
```

Uma função de comparação, neste caso, receberia como parâmetros dois ponteiros para a estrutura que representa um aluno e, considerando uma ordenação que usa o nome do aluno como chave de comparação, poderia ter a seguinte implementação:

```
int compara (Aluno* a, Aluno* b)
{
    if (strcmp(a->nome,b->nome) > 0)
        return 1;
    else
        return 0;
}
```

Portanto, o uso de uma função auxiliar para realizar a comparação entre os elementos ajuda para a obtenção de um código reusável. No entanto, isto só não é suficiente. Para o mesmo código poder ser aplicado a qualquer tipo de informação armazenada no vetor, precisamos tornar a implementação independente do tipo do elemento, isto é, precisamos tornar tanto a própria função de ordenação (`bolha`) quanto a função de comparação (`compara`) independentes do tipo do elemento.

Em C, a forma de generalizar o tipo é usar o tipo `void*`. Escreveremos o código de ordenação considerando que temos um ponteiro de qualquer tipo e passaremos para a função de comparação dois ponteiros genéricos, um para cada elemento que se deseja comparar. A função de ordenação, no entanto, precisa percorrer o vetor e para tanto precisamos passar para a função uma informação adicional—o tamanho, em número de bytes, de cada elemento. A assinatura da função de ordenação poderia então ser dada por:

```
void bubble (int n, void* v, int tam);
```

A função de ordenação por sua vez, recebe dois ponteiros genéricos:

```
int compara (void* a, void* b);
```

Assim, se estamos ordenando vetores de inteiros, escrevemos a nossa função de comparação convertendo o ponteiro genérico para um ponteiro de inteiro e fazendo o teste apropriado:

```

/* função de comparação para inteiros */
int compara (void* a, void* b)
{
    int* p1 = (int*) a;
    int* p2 = (int*) b;
    int i1 = *p1;
    int i2 = *p2;
    if (i1 > i2)
        return 1;
    else
        return 0;
}

```

Se os elementos do vetor fossem ponteiros para a estrutura aluno, a função de comparação poderia ser:

```

/* função de comparação para ponteiros de alunos */
int compara (void* a, void* b)
{
    Aluno** p1 = (Aluno**) a;
    Aluno** p2 = (Aluno**) b;
    Aluno* i1 = *p1;
    Aluno* i2 = *p2;
    if (strcmp(i1->nome, i2->nome) > 0)
        return 1;
    else
        return 0;
}

```

O código da função de ordenação necessita percorrer os elementos do vetor. O acesso a um determinado elemento i do vetor não pode mais ser feito diretamente por $v[i]$. Dado o endereço do primeiro elemento do vetor, devemos incrementar este endereço de $i*tam$ bytes para termos o endereço do elemento i . Podemos então escrever uma função auxiliar que faz esse incremento de endereço. Essa função recebe como parâmetros o endereço inicial do vetor, o índice do elemento cujo endereço se quer alcançar e o tamanho (em bytes) de cada elemento. A função retorna o endereço do elemento especificado. Uma parte útil, porém necessária, dessa função é que para incrementar o endereço genérico de um determinado número de bytes, precisamos antes, temporariamente, converter esse ponteiro para ponteiro para caractere (pois um caractere ocupa um byte). O código dessa função auxiliar pode ser dado por:

```

void* acessa (void* v, int i, int tam)
{
    char* t = (char*)v;
    t += tam*i;
    return (void*)t;
}

```

A função de ordenação identifica a ocorrência de inversões entre elementos e realiza uma troca entre os valores. O código que realiza a troca também tem que ser pensado de forma genérica, pois, como não sabemos o tipo de cada elemento, não temos como declarar a variável temporária para poder realizar a troca. Uma alternativa é fazer a troca dos valores byte a byte (ou caractere a caractere). Para tanto, podemos definir uma outra função auxiliar que recebe os ponteiros genéricos dos dois elementos que devem ter seus valores trocados, além do tamanho de cada elemento.

```

void troca (void* a, void* b, int tam)
{
    char* v1 = (char*) a;
    char* v2 = (char*) b;
    int i;
    for (i=0; i<tam; i++) {
        char temp = v1[i];
        v1[i] = v2[i];
        v2[i] = temp;
    }
}

```

Assim, podemos escrever o código da nossa função de ordenação genérica. Falta, no entanto, um último detalhe. As funções auxiliares `acessa` e `troca` são realmente genéricas, e independem da informação efetivamente armazenada no vetor. Porém, a função de comparação deve ser especializada para cada tipo de informação, conforme ilustramos acima. A assinatura dessa função é genérica, mas a sua implementação deve, naturalmente, levar em conta a informação armazenada para que a comparação tenha sentido. Portanto, para generalizar a implementação da função de ordenação, não podemos chamar uma função de comparação específica. A solução é passar, via parâmetro, qual função de ordenação deve ser chamada. Para tanto, temos que introduzir o conceito de *ponteiro para função*. O nome de uma função representa o endereço dessa função. A nossa função de comparação tem a assinatura:

```
int compara (void*, void*);
```

Uma variável ponteiro para armazenar o endereço dessa função é declarada como:

```
int (*cmp) (void*,void*);
```

onde `cmp` representa a variável do tipo ponteiro para a função em questão.

Agora sim, podemos escrever nossa função de ordenação genérica, recebendo como parâmetro adicional o ponteiro da função de comparação:

```

/* Ordenação bolha (genérica) */
void bolha_gen (int n, void* v, int tam, int (*cmp) (void*,void*))
{
    int i, j;
    for (i=n-1; i>0; i--) {
        int fez_troca = 0;
        for (j=0; j<i; j++) {
            void* p1 = acessa(v,j,tam);
            void* p2 = acessa(v,j+1,tam);
            if (cmp(p1,p2))
                troca(p1,p2,tam);
            fez_troca = 1;
        }
        if (fez_troca == 0) /* nao houve troca */
            return;
    }
}

```

Esse código genérico pode ser usado para ordenar vetores com qualquer informação. Para exemplificar, vamos usá-lo para ordenar um vetor de números reais. Para isso, temos que escrever o código da função que faz a comparação, agora especializada para número reais:

```

int compara_reais (void* a, void* b)
{
    float* p1 = (float*) a;
    float* p2 = (float*) b;
    float f1 = *p1;
    float f2 = *p2;
    if (f1 > f2)
        return 1;
    else
        return 0;
}

```

Podemos, então, chamar a função para ordenar um vetor v de n números reais:

```

...
bubble_gen(n,v,sizeof(float),compara_reais);
...

```

15.2. Ordenação Rápida

Assim como o algoritmo anterior, o algoritmo “ordenação rápida”, “*quick sort*”, que iremos discutir agora, procura resolver o problema da ordenação por partes. No entanto, enquanto o algoritmo de ordenação bolha coloca em sua posição (no final do vetor) o maior elemento, a ordenação rápida faz isso com um elemento arbitrário x , chamado de pivô. Por exemplo, podemos escolher como pivô o primeiro elemento do vetor, e posicionar esse elemento em sua correta posição numa primeira passada.

Suponha que este elemento, x , deva ocupar a posição i do vetor, de acordo com a ordenação, ou seja, que essa seja a sua posição definitiva no vetor. Sem ordenar o vetor completamente, este fato pode ser reconhecido quando todos os elementos $v[0], \dots, v[i-1]$ são menores que x , e todos os elementos $v[i+1], \dots, v[n-1]$ são maiores que x . Supondo que x já está na sua posição correta, com índice i , há dois problemas menores para serem resolvidos: ordenar os (sub-) vetores formados por $v[0], \dots, v[i-1]$ e por $v[i+1], \dots, v[n-1]$. Esses sub-problemas são resolvidos (recursivamente) de forma semelhante, cada vez com vetores menores, e o processo continua até que os vetores que devem ser ordenados tenham zero ou um elementos, caso em que sua ordenação já está concluída.

A grande vantagem desse algoritmo é que ele pode ser muito eficiente. O melhor caso ocorre quando o elemento pivô representa o valor mediano do conjunto dos elementos do vetor. Se isto acontece, após o posicionamento do pivô em sua posição, restará dois sub-vetores para serem ordenados, ambos com o número de elementos reduzido a metade, em relação ao vetor original. Pode-se mostrar que, neste melhor caso, o esforço computacional do algoritmo é proporcional a $n \log(n)$, e dizemos que o algoritmo é $O(n \log(n))$. Um desempenho muito superior ao $O(n^2)$ apresentado pelo algoritmo de ordenação bolha. Infelizmente, não temos como garantir que o pivô seja o mediano. No pior caso, o pivô pode sempre ser, por exemplo, o maior elemento, e recaímos no algoritmo de ordenação bolha. No entanto, mostra-se que o algoritmo *quicksort* ainda apresenta, no caso médio, um desempenho $O(n \log(n))$.

A versão do “*quicksort*” que vamos apresentar aqui usa $x=v[0]$ como primeiro elemento a ser colocado em sua posição correta. O processo compara os elementos $v[1], v[2], \dots$ até encontrar um elemento $v[a] > x$. Então, a partir do final do vetor, compara os elementos $v[n-1], v[n-2], \dots$ até encontrar um elemento $v[b] <= x$.

Neste ponto, $v[a]$ e $v[b]$ são trocados, e a busca continua, para cima a partir de $v[a+1]$, e para baixo, a partir de $v[b-1]$. Em algum momento, a busca termina, porque os pontos de busca se encontrarão. Neste momento, a posição correta de x está definida, e os valores $v[0]$ e $v[a]$ são trocados.

Vamos usar o mesmo exemplo da seção anterior:

(0-7) 25 48 37 12 57 86 33 92

onde indicamos através de (0-7) que se trata do vetor inteiro, de $v[0]$ a $v[7]$. Podemos começar a executar o algoritmo procurando determinar a posição correta de $x=v[0]=25$. Partindo do início do vetor, já temos, na primeira comparação, $48 > 25$ ($a=1$). Partindo do final do vetor, na direção oposta, temos $25 < 92$, $25 < 33$, $25 < 86$, $25 < 57$ e finalmente, $12 <= 25$ ($b=3$).

(0-7) 25 48 37 12 57 86 33 92
 $a \uparrow$ $b \uparrow$

Trocamos então $v[a]=48$ e $v[b]=12$, incrementando a de uma unidade e decrementando b de uma unidade. Os elementos do vetor ficam com a seguinte disposição:

(0-7) 25 12 37 48 57 86 33 92
 $a, b \uparrow$

Na continuação, temos $37 > 25$ ($a=2$). Pelo outro lado, chegamos também a 37 e temos $37 > 25$ e $12 <= 25$. Neste ponto, verificamos que os índices a e b se cruzaram, agora com $b < a$.

(0-7) 25 12 37 48 57 86 33 92
 $b \uparrow$ $a \uparrow$

Assim, todos os elementos de 37 (inclusive) em diante são maiores que 25, e todos os elementos de 12 (inclusive) para trás são menores que 25. Com exceção do próprio 25, é claro. A próxima etapa troca o pivô, $v[0]=25$, com o último dos valores menores que 25 encontrado: $v[b]=12$. Temos:

(0-7) 12 25 37 48 57 86 33 92

com 25 em sua posição correta, e dois vetores menores para ordenar. Valores menores que 25:

(0-0) 12

E valores maiores:

(2-7) 37 48 57 86 33 92

Neste caso, em particular, o primeiro vetor (com apenas um elemento: (0-0)) já se encontra ordenado. O segundo vetor (2-7) pode ser ordenado de forma semelhante:

(2-7) 37 48 57 86 33 92

Devemos decidir qual a posição correta de 37. Para isso identificamos o primeiro elemento maior que 37, ou seja, 48, e o último menor que 37, ou seja, 33.

(2-7) 37 48 57 86 33 92
a↑ b↑

Trocamos os elementos e atualizamos os índices:

(2-7) 37 33 57 86 48 92
a↑ b↑

Continuando o processo, verificamos que $37 < 57$ e $37 < 86$, $37 < 57$, mas $37 >= 33$. Identificamos novamente que a e b se cruzaram.

(2-7) 37 33 57 86 48 92
b↑ a↑

Assim, a posição correta de 37 é a posição ocupada por $v[b]$, e os dois elementos devem ser trocados:

(2-7) 33 37 57 86 48 92

restando os vetores

(2-2) 33

e

(4-7) 57 86 48 92

para serem ordenados.

O processo continua até que o vetor original esteja totalmente ordenado.

(0-7) 12 25 33 37 48 57 86 92

A implementação do *quick sort* é normalmente recursiva, para facilitar a ordenação dos dois vetores menores encontrados. A seguir, apresentamos uma possível implementação do algoritmo, adotando como pivô o primeiro elemento.

```

/* Ordenação rápida */
void rapida (int n, int* v)
{
    if (n <= 1)
        return;
    else {
        int x = v[0];
        int a = 1;
        int b = n-1;
        do {
            while (a < n && v[a] <= x) a++;
            while (v[b] > x) b--;
            if (a < b) { /* faz troca */
                int temp = v[a];
                v[a] = v[b];
                v[b] = temp;
                a++; b--;
            }
        } while (a <= b);

        /* troca pivô */
        v[0] = v[b];
        v[b] = x;

        /* ordena sub-vetores restantes */
        rapida(b, v);
        rapida(n-a, &v[a]);
    }
}

```

Devemos observar que para deslocar o índice *a* para a direita, fizemos o teste:

```
while (a < n && v[a] <= x)
```

enquanto que para deslocar o índice *b* para a esquerda, fizemos apenas:

```
while (v[b] > x)
```

O teste adicional no deslocamento para a direita é necessário porque o pivô pode ser o elemento de maior valor, nunca ocorrendo a situação $v[a] \leq x$, o que nos faria acessar posições além dos limites do vetor. No deslocamento para a esquerda, um teste adicional tipo $b \geq 0$ não é necessário, pois, na nossa implementação, $v[0]$ é o pivô, impedindo que *b* assuma valores negativos (teremos, pelo menos, $x \geq v[0]$).

Algoritmo genérico da biblioteca padrão

O *quicksort* é o algoritmo de ordenação mais utilizado no desenvolvimento de aplicações. Mesmo quando temos os dados organizados em listas encadeadas, e precisamos colocá-los de forma ordenada, em geral, optamos por criar um vetor temporário com ponteiros para os nós da lista, fazer a ordenação usando *quicksort* e reencadear os nós montando a lista ordenada.

Devido a sua grande utilidade, a biblioteca padrão de C disponibiliza, via a interface `stdlib.h`, uma função que ordena vetores usando esse algoritmo. A função disponibilizada pela biblioteca independe do tipo de informação armazenada no vetor. A implementação dessa função genérica segue os princípios discutidos na implementação do algoritmo de ordenação bolha genérico. O protótipo da função disponibilizada pela biblioteca é:

```
void qsort (void *v, int n, int tam, int (*cmp)(const void*, const void*));
```

Os parâmetros de entrada dessa função são:

v: o ponteiro para o primeiro elemento do vetor que se deseja ordenar. Como não se sabe, a priori, o tipo dos elementos do vetor, temos um ponteiro genérico – `void*`.

n: o número de elementos do vetor.

tam: o tamanho, em bytes, de cada elemento do vetor.

cmp: o ponteiro para a função responsável por comparar dois elementos do vetor. Em C, o nome de uma função representa o *ponteiro da função*. Esse ponteiro pode ser armazenado numa variável, possibilitando chamar a função indiretamente. Como era de se esperar, a biblioteca não sabe comparar dois elementos do vetor (ela desconhece o tipo desses elementos). Fica a cargo do cliente da função de ordenação escrever a função de comparação. Essa função de comparação tem que ter o seguinte protótipo:

```
int nome (const void*, const void*);
```

O parâmetro `cmp` recebido pela função `qsort` é um ponteiro para uma função com esse protótipo. Assim, para usarmos a função de ordenação da biblioteca temos que escrever uma função que receba dois ponteiros genéricos, `void*`, os quais representam ponteiros para os dois elementos que se deseja comparar. O modificador de tipo `const` aparece no protótipo apenas para garantir que essa função não modificará os valores dos elementos (devem ser tratados como valores constantes). Essa função deve ter como valor de retorno -1, 0, ou 1, dependendo se o primeiro elemento for menor, igual, ou maior que o segundo, respectivamente.

Para ilustrar a utilização da função `qsort` vamos considerar alguns exemplos. O código a seguir ilustra a utilização da função para ordenar valores reais. Neste caso, os dois ponteiros genéricos passados para a função de comparação representam ponteiros para `float`.

```
/* Ilustra uso do algoritmo qsort */
#include <stdio.h>
#include <stdlib.h>

/* função de comparação de reais */
int comp_reais (const void* p1, const void* p2)
{
    /* converte ponteiros genéricos para ponteiros de float */
    float *f1 = (float*)p1;
    float *f2 = (float*)p2;
    /* dados os ponteiros de float, faz a comparação */
    if (*f1 < *f2) return -1;
    else if (*f1 > *f2) return 1;
    else return 0;
}
```

```

/* programa que faz a ordenação de um vetor */
int main (void)
{
    int i;
    float v[8] = {25.6,48.3,37.7,12.1,57.4,86.6,33.3,92.8};

    qsort(v,8,sizeof(float),comp_reais);

    printf("Vetor ordenado: ");
    for (i=0; i<8; i++)
        printf("%g ",v[i]);
    printf("\n");
    return 0;
}

```

Vamos agora considerar que temos um vetor de alunos e que desejamos ordenar o vetor usando o nome do aluno como chave de comparação. A estrutura que representa um aluno pode ser dada por:

```

struct aluno {
    char nome[81];
    char mat[8];
    char turma;
    char email[41];
};
typedef struct aluno Aluno;

```

Vamos analisar duas situações. Na primeira, consideraremos a existência de um vetor da estrutura (por exemplo, `Aluno vet[N];`). Neste caso, cada elemento do vetor é do tipo `Aluno` e os dois ponteiros genéricos passados para a função de comparação representam ponteiros para `Aluno`. Essa função de comparação pode ser dada por:

```

/* Função de comparação: elemento é do tipo Aluno */
int comp_alunos (const void* p1, const void* p2)
    /* converte ponteiros genéricos para ponteiros de Aluno */
    Aluno *a1 = (Aluno*)p1;
    Aluno *a2 = (Aluno*)p2;
    /* dados os ponteiros de Aluno, faz a comparação */
    return strcmp(a1->nome,a2->nome);
}

```

Numa segunda situação, podemos considerar que temos um vetor de ponteiros para a estrutura `aluno` (por exemplo, `Aluno* vet[N];`). Agora, cada elemento do vetor é um ponteiro para o tipo `Aluno` e a função de comparação tem que tratar uma indireção a mais. Aqui, os dois ponteiros genéricos passados para a função de comparação representam ponteiros de ponteiros para `Aluno`.

```

/* Função de comparação: elemento é do tipo Aluno* */
int comp_alunos (const void* p1, const void* p2)
    /* converte ponteiros genéricos para ponteiros de ponteiros de Aluno */
    Aluno **pa1 = (Aluno**)p1;
    Aluno **pa2 = (Aluno**)p2;
    /* acessa ponteiro de Aluno */
    Aluno *a1 = *p1;
    Aluno *a2 = *p2;
    /* dados os ponteiros de Aluno, faz a comparação */
    return strcmp(a1->nome,a2->nome);
}

```

16. Busca

W. Celes e J. L. Rangel

Neste capítulo, discutiremos diferentes estratégias para efetuarmos a busca de um elemento num determinado conjunto de dados. A operação de busca é encontrada com muita freqüência em aplicações computacionais, sendo portanto importante estudar estratégias distintas para efetuá-la. Por exemplo, um programa de controle de estoque pode buscar, dado um código numérico ou um nome, a descrição e as características de um determinado produto. Se temos um grande número de produtos cadastrados, o método para efetuar a busca deve ser eficiente, caso contrário a busca pode ser muito demorada, inviabilizando sua utilização.

Neste capítulo, estudaremos algumas estratégias de busca. Inicialmente, consideraremos que temos nossos dados armazenados em um vetor e discutiremos os algoritmos de busca que podemos empregar. A seguir, discutiremos a utilização de árvores binárias de busca, que são estruturas de árvores projetadas para darem suporte a operações de busca de forma eficiente. No próximo capítulo, discutiremos as estruturas conhecidas como tabelas de dispersão (*hash*) que podem, como veremos, realizar buscas de forma extremamente eficiente, fazendo uso de espaço de memória adicional.

16.1. Busca em Vetor

Nesta seção, apresentaremos os algoritmos de busca em vetor. Dado um vetor `vet` com n elementos, desejamos saber se um determinado elemento `elem` está ou não presente no vetor.

Busca linear

A forma mais simples de fazermos uma busca num vetor consiste em percorrermos o vetor, elemento a elemento, verificando se o elemento de interesse é igual a um dos elementos do vetor. Esse algoritmo pode ser implementado conforme ilustrado pelo código a seguir, considerando-se um vetor de números inteiros. A função apresentada tem como valor de retorno o índice do vetor no qual foi encontrado o elemento; se o elemento não for encontrado, o valor de retorno é -1 .

```
int busca (int n, int* vet, int elem)
{
    int i;

    for (i=0; i<n; i++) {
        if (elem == vet[i])
            return i;      /* elemento encontrado */
    }

    /* percorreu todo o vetor e não encontrou elemento */
    return -1;
}
```

Esse algoritmo de busca é extremamente simples, mas pode ser muito ineficiente quando o número de elementos no vetor for muito grande. Isto porque o algoritmo (a função, no caso) pode ter que percorrer todos os elementos do vetor para verificar que um determinado elemento está ou não presente. Dizemos que no pior caso será necessário realizar n comparações, onde n representa o número de elementos no vetor.

Portanto, o desempenho computacional desse algoritmo varia linearmente com relação ao tamanho do problema – chamamos esse algoritmo de *busca linear*.

Em geral, usamos a notação “Big-O” para expressarmos como a complexidade de um algoritmo varia com o tamanho do problema. Assim, nesse caso em que o tempo computacional varia linearmente com o tamanho do problema, dizemos que trata-se de um algoritmo de ordem linear e expressamos isto escrevendo $O(n)$.

No melhor caso, se dermos sorte do elemento procurado ocupar a primeira posição do vetor, o algoritmo acima necessitaria de apenas uma única comparação. Esse fato, no entanto, não pode ser usado para fazermos uma análise de desempenho do algoritmo, pois o melhor caso representa um situação muito particular.

Além do pior caso, devemos analisar o caso médio, isto é, o caso que ocorre na média. Já vimos que o algoritmo em questão requer n comparações quando o elemento não está presente no vetor. E no caso do elemento estar presente, quantas operações de comparação são, em média, necessárias? Na média, podemos concluir que são necessárias $n/2$ comparações. Em termos de ordem de complexidade, no entanto, continuamos a ter uma variação linear, isto é, $O(n)$, pois dizemos que $O(k n)$, onde k é uma constante, é igual a $O(n)$.

Em diversas aplicações reais, precisamos de algoritmos de busca mais eficientes. Seria possível melhorarmos a eficiência do algoritmo de busca mostrado acima? Infelizmente, se os elementos estiverem armazenados em uma ordem aleatória no vetor, não temos como melhorar o algoritmo de busca, pois precisamos verificar todos os elementos. No entanto, se assumirmos, por exemplo, que os elementos estão armazenados em ordem crescente, podemos concluir que um elemento não está presente no vetor se acharmos um elemento maior, pois se o elemento que buscamos estivesse presente ele precederia um elemento maior na ordem do vetor.

O código abaixo ilustra a implementação da busca linear assumindo que os elementos do vetor estão ordenados (vamos assumir ordem crescente).

```
int busca_ord (int n, int* vet, int elem)
{
    int i;

    for (i=0; i<n; i++) {
        if (elem == vet[i])
            return i; /* elemento encontrado */
        else if (elem < vet[i])
            return -1; /* interrompe busca */
    }

    /* percorreu todo o vetor e não encontrou elemento */
    return -1;
}
```

No caso do elemento procurado não pertencer ao vetor, esse segundo algoritmo apresenta um desempenho ligeiramente superior ao primeiro, mas a ordem dessa versão do algoritmo continua sendo linear – $O(n)$. No entanto, se os elementos do vetor estão ordenados, existe um algoritmo muito mais eficiente que será apresentado a seguir.

Busca binária

No caso dos elementos do vetor estarem em ordem, podemos aplicar um algoritmo mais eficiente para realizarmos a busca. Trata-se do algoritmo de *busca binária*. A idéia do algoritmo é testar o elemento que buscamos com o valor do elemento armazenado no meio do vetor. Se o elemento que buscamos for menor que o elemento do meio, sabemos que, se o elemento estiver presente no vetor, ele estará na primeira parte do vetor; se for maior, estará na segunda parte do vetor; se for igual, achamos o elemento no vetor. Se concluirmos que o elemento está numa das partes do vetor, repetimos o procedimento considerando apenas a parte que restou: comparamos o elemento que buscamos com o elemento armazenado no meio dessa parte. Este procedimento é continuamente repetido, subdividindo a parte de interesse, até encontrarmos o elemento ou chegarmos a uma parte do vetor com tamanho zero.

O código a seguir ilustra uma implementação de busca binária num vetor de valores inteiros ordenados de forma crescente.

```
int busca_bin (int n, int* vet, int elem)
{
    /* no inicio consideramos todo o vetor */
    int ini = 0;
    int fim = n-1;
    int meio;

    /* enquanto a parte restante for maior que zero */
    while (ini <= fim) {
        meio = (ini + fim) / 2;
        if (elem < vet[meio])
            fim = meio - 1;           /* ajusta posição final */
        else if (elem > vet[meio])
            ini = meio + 1;         /* ajusta posição inicial */
        else
            return meio;           /* elemento encontrado */
    }

    /* não encontrou: restou parte de tamanho zero */
    return -1;
}
```

O desempenho desse algoritmo é muito superior ao de busca linear. Novamente, o pior caso caracteriza-se pela situação do elemento que buscamos não estar no vetor. Quantas vezes precisamos repetir o procedimento de subdivisão para concluirmos que o elemento não está presente no vetor? A cada repetição, a parte considerada na busca é dividida à metade. A tabela abaixo mostra o tamanho do vetor a cada repetição do laço do algoritmo.

Repetição	Tamanho do problema
1	n
2	$n/2$
3	$n/4$
...	...
$\log n$	1

Sendo assim necessárias $\log n$ repetições. Como fazemos um número constante de comparações a cada ciclo (duas comparações por ciclo), podemos concluir que a ordem desse algoritmo é $O(\log n)$.

O algoritmo de busca binária consiste em repetirmos o mesmo procedimento recursivamente, podendo ser naturalmente implementado de forma recursiva. Embora a implementação não recursiva seja mais eficiente e mais adequada para esse algoritmo, a implementação recursiva é mais sucinta e vale a pena ser apresentada. Na implementação recursiva, temos dois casos a serem tratados. No primeiro, a busca deve continuar na primeira metade do vetor, logo chamamos a função recursivamente passando como parâmetros o número de elementos dessa primeira parte restante e o mesmo ponteiro para o primeiro elemento, pois a primeira parte tem o mesmo primeiro elemento do que o vetor como um todo. No segundo caso, a busca deve continuar apenas na segunda parte do vetor, logo passamos na chamada recursiva, além do número de elementos restantes, um ponteiro para o primeiro elemento dessa segunda parte. Para simplificar, a função de busca apenas informa se o elemento pertence ou não ao vetor, tendo como valor de retorno falso (0) ou verdadeiro (1). Uma possível implementação usando essa estratégia é mostrada a seguir.

```
int busca_bin_rec (int n, int* vet, int elem)
{
    /* testa condição de contorno: parte com tamanho zero */
    if (n <= 0)
        return 0;
    else {
        /* deve buscar o elemento entre os índices 0 e n-1 */
        int meio = (n - 1) / 2;

        if (elem < vet[meio])
            return busca_bin_rec(meio, vet, elem);
        else if (elem > vet[meio])
            return busca_bin_rec(n-1-meio, &vet[meio+1], elem);
        else
            return 1;           /* elemento encontrado */
    }
}
```

Em particular, devemos notar a expressão `&vet[meio+1]` que, como sabemos, resulta num ponteiro para o primeiro elemento da segunda parte do vetor.

Se quisermos que a função tenha como valor de retorno o índice do elemento, devemos acertar o valor retornado pela chamada recursiva na segunda parte do vetor. Uma implementação com essa modificação é apresentada abaixo:

```
int busca_bin_rec (int n, int* vet, int elem)
{
    /* testa condição de contorno: parte com tamanho zero */
    if (n <= 0)
        return -1;
    else {
        /* deve buscar o elemento entre os índices 0 e n-1 */
        int meio = (n - 1) / 2;

        if (elem < vet[meio])
            return busca_bin_rec(meio, vet, elem);
        else if (elem > vet[meio])
        {
            int r = busca_bin_rec(n-1-meio, &vet[meio+1], elem);
            if (r<0) return -1;
            else      return meio+1+r;
        }
        else
            return meio;           /* elemento encontrado */
    }
}
```

Algoritmo genérico

A biblioteca padrão de C disponibiliza, via a interface `stdlib.h`, uma função que faz a busca binária de um elemento num vetor. A função disponibilizada pela biblioteca independe do tipo de informação armazenada no vetor. A implementação dessa função genérica segue os mesmos princípios discutidos no capítulo anterior. O protótipo da função de busca binária da biblioteca é:

```
void* bsearch (void* info, void *v, int n, int tam,
               int (*cmp) (const void*, const void*)
               );
```

Se o elemento for encontrado no vetor, a função tem como valor de retorno o endereço do elemento no vetor; caso o elemento não seja encontrado, o valor de retorno é `NULL`. Análogo a função `qsort`, apresentada no capítulo anterior, os parâmetros de entrada dessa função são:

- info: o ponteiro para a informação que se deseja buscar no vetor – representa a *chave de busca*;
- v: o ponteiro para o primeiro elemento do vetor onde a busca será feita. Os elementos do vetor têm que estar ordenados, segundo o critério de ordenação adotado pela função de comparação descrita abaixo.
- n: o número de elementos do vetor.
- tam: o tamanho, em bytes, de cada elemento do vetor.
- cmp: o ponteiro para a função responsável por comparar a informação buscada e um elemento do vetor. O primeiro parâmetro dessa função é sempre o endereço da informação buscada, e o segundo é um ponteiro para um dos elementos do vetor. O critério de comparação adotado por essa função deve ser compatível com o critério de ordenação do vetor. Essa função deve ter como valor de retorno `-1`, `0`, ou `1`, dependendo se a informação buscada for menor, igual, ou maior que a informação armazenada no elemento, respectivamente.

Para ilustrar a utilização da função `bsearch` vamos, inicialmente, considerar um vetor de valores inteiros. Neste caso, os dois ponteiros genéricos passados para a função de comparação representam ponteiros para `int`.

```
/* Ilustra uso do algoritmo bsearch */
#include <stdio.h>
#include <stdlib.h>

/* função de comparação de inteiros */
int comp_int (const void* p1, const void* p2)
{
    /* converte ponteiros genéricos para ponteiros de int */
    int *i1 = (int*)p1;
    int *i2 = (int*)p2;
    /* dados os ponteiros de int, faz a comparação */
    if (*i1 < *i2) return -1;
    else if (*i1 > *i2) return 1;
    else return 0;
}
```

```

/* programa que faz a busca em um vetor */
int main (void)
{
    int v[8] = {12,25,33,37,48,57,86,92};
    int e = 57;      /* informação que se deseja buscar */
    int* p;

    p = (int*)bsearch(&e,v,8,sizeof(int),comp_int);

    if (p == NULL)
        printf("Elemento nao encontrado.\n");
    else
        printf("Elemento encontrado no indice: %d\n", p-v);
    return 0;
}

```

Devemos notar que o índice do elemento, se encontrado no vetor, pode ser extraído subtraindo-se o ponteiro do elemento do ponteiro do primeiro elemento ($p - v$). Essa *aritmética de ponteiros* é válida aqui pois podemos garantir que ambos os ponteiros armazenam endereços de memória de um mesmo vetor. A diferença entre os ponteiros representa a “distância” em que os elementos estão armazenados na memória.

Vamos agora considerar que queremos efetuar uma busca num vetor de ponteiros para alunos. A estrutura que representa um aluno pode ser dada por:

```

struct aluno {
    char nome[81];
    char mat[8];
    char turma;
    char email[41];
};

typedef struct aluno Aluno;

```

Considerando que o vetor está ordenado segundo os nomes dos alunos, podemos buscar a ocorrência de um determinado aluno passando para a função de busca um nome e o vetor. A função de comparação então receberá dois ponteiros: um ponteiro para uma cadeia de caracteres e um ponteiro para um elemento do vetor (no caso será um ponteiro para ponteiro de aluno, ou seja, um `Aluno**`).

```

/* Função de comparação: char* e Aluno** */
int comp_alunos (const void* p2, const void* p1)
    /* converte ponteiros genéricos para ponteiros específicos */
    char* s = (char*)p1;
    Aluno **pa = (Aluno**)p2;
    /* faz a comparação */
    return strcmp(s,(*pa)->nome);
}

```

Conforme observamos, o tipo de informação a ser buscada nem sempre é igual ao tipo do elemento; para dados complexos, em geral não é. A informação buscada geralmente representa um campo da estrutura armazenada no vetor (ou da estrutura apontada por elementos do vetor).

Devemos finalmente salientar que se tivermos os dados armazenados em uma lista encadeada, só temos a alternativa de implementar um algoritmo de busca linear, mesmo se os elementos estiverem ordenados. Portanto, lista encadeada não é uma boa opção para estruturarmos nossos dados, se desejarmos realizar muitas operações de busca. A estrutura dinâmica apropriada para a realização de busca é a árvore binária de busca que será discutida a seguir.

16.2. Árvore binária de busca

Como vimos, o algoritmo de busca binária apresentado na seção anterior apresenta bom desempenho computacional e deve ser usado quando temos os dados ordenados armazenados num vetor. No entanto, se precisarmos inserir e remover elementos da estrutura, e ao mesmo tempo dar suporte a eficientes funções de busca, a estrutura de vetor (e, consequentemente, o uso do algoritmo de busca binária) não se torna adequada. Para inserirmos um novo elemento num vetor ordenado, temos que rearranjar os elementos no vetor, para abrir espaço para a inserção do novo elemento. Situação análoga ocorre quando removemos um elemento do vetor. Precisamos portanto de uma estrutura dinâmica que dê suporte a operações de busca.

Um dos resultados que apresentamos anteriormente foi o da relação entre o número de nós de uma árvore binária e sua altura. A cada nível, o número (potencial) de nós vai dobrando, de maneira que uma árvore binária de altura h pode ter um número de nós dado por:

$$1 + 2 + 2^2 + \dots + 2^{h-1} + 2^h = 2^{h+1} - 1$$

Assim, dizemos que uma árvore binária de altura h pode ter no máximo $O(2^h)$ nós, ou, pelo outro lado, que uma árvore binária com n nós pode ter uma altura mínima de $O(\log n)$. Essa relação entre o número de nós e a altura mínima da árvore é importante porque se as condições forem favoráveis, podemos alcançar qualquer um dos n nós de uma árvore binária a partir da raiz em, no máximo, $O(\log n)$ passos. Se tivéssemos os n nós em uma lista linear, o número máximo de passos seria $O(n)$, e, para os valores de n encontrados na prática, $\log n$ é muito menor do que n .

A altura de uma árvore é, certamente, uma medida do tempo necessário para encontrar um dado nó. No entanto, é importante observar que para acessarmos qualquer nó de maneira eficiente é necessário termos árvores binárias “balanceadas”, em que os nós internos têm todos, ou quase todos, o máximo número de filhos, no caso 2. Lembramos que o número mínimo de nós de uma árvore binária de altura h é $h+1$, de forma que a altura máxima de uma árvore com n nós é $O(n)$. Esse caso extremo corresponde à árvore “degenerada”, em que todos os nós têm apenas 1 filho, com exceção da (única) folha.

As árvores binárias que serão consideradas nesta seção têm uma propriedade fundamental: o valor associado à raiz é sempre maior que o valor associado a qualquer nó da sub-árvore à esquerda (*sae*), e é sempre menor que o valor associado a qualquer nó da sub-árvore à direita (*sad*). Essa propriedade garante que, quando a árvore é percorrida em ordem simétrica (*sae - raiz - sad*), os valores são encontrados em ordem crescente.

Uma variação possível permite que haja repetição de valores na árvore: o valor associado à raiz é sempre maior que o valor associado a qualquer nó da *sae*, e é sempre menor ou igual ao valor associado a qualquer nó da *sad*. Nesse caso, como a repetição de valores é permitida, quando a árvore é percorrida em ordem simétrica, os valores são encontrados em ordem não decrescente.

Usando essa propriedade de ordem, a busca de um valor em uma árvore pode ser simplificada. Para procurar um valor numa árvore, comparamos o valor que buscamos com o valor associado à raiz. Em caso de igualdade, o valor foi encontrado; se o valor dado for menor que o valor associado à raiz, a busca continua na *sae*; caso contrário, se o valor associado à raiz for menor, a busca continua na *sad*. Por essa razão, estas árvores são freqüentemente chamadas de *árvores binárias de busca*.

Naturalmente, a ordem a que fizemos referência acima é dependente da aplicação. Se a informação a ser armazenada em cada nó da árvore for um número inteiro podemos usar o habitual operador relacional *menor que* (“<”). Porém, se tivermos que considerar casos em que a informação é mais complexa, já vimos que uma função de comparação deve ser definida pelo programador, especificamente para cada caso.

Operações em árvores binárias de busca

Para exemplificar a implementação de operações em árvores binárias de busca, vamos considerar o caso em que a informação associada a um nó é um número inteiro, e não vamos considerar a possibilidade de repetição de valores associados aos nós da árvore. A Figura 16.1 ilustra uma árvore de busca de valores inteiros.

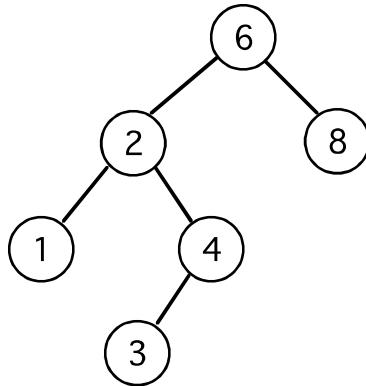


Figura 16.1: Exemplo de árvore binária de busca.

O tipo da árvore binária pode então ser dado por:

```

struct arb {
    int info;
    struct arb* esq;
    struct arb* dir;
};

typedef struct arb Arv;
  
```

A árvore é representada pelo ponteiro para o nó raiz. A árvore vazia é inicializada atribuindo-se `NULL` a variável que representa a árvore. Uma função simples de inicialização é mostrada abaixo:

```

Arv* init (void)
{
    return NULL;
}
  
```

Uma vez construída uma árvore de busca, podemos imprimir os valores da árvore em ordem crescente percorrendo os nós em ordem simétrica:

```
void imprime (Arv* a)
{
    if (a != NULL) {
        imprime(a->esq);
        printf("%d\n", a->info);
        imprime(a->dir);
    }
}
```

Essas são funções análogas às vistas para árvores binárias comuns, pois não exploram a propriedade de ordenação das árvores de busca. No entanto, as operações que nos interessa analisar em detalhes são:

- busca**: função que busca um elemento na árvore;
- insere**: função que insere um novo elemento na árvore;
- retira**: função que retira um elemento da árvore.

Operação de busca

A operação para buscar um elemento na árvore explora a propriedade de ordenação da árvore, tendo um desempenho computacional proporcional a sua altura ($O(\log n)$ para o caso de árvore balanceada). Uma implementação da função de busca é dada por:

```
Arv* busca (Arv* r, int v)
{
    if (r == NULL) return NULL;
    else if (r->info > v) return busca (r->esq, v);
    else if (r->info < v) return busca (r->dir, v);
    else return r;
}
```

Operação de inserção

A operação de inserção adiciona um elemento na árvore na posição correta para que a propriedade fundamental seja mantida. Para inserir um valor v em uma árvore usamos sua estrutura recursiva, e a ordenação especificada na propriedade fundamental. Se a (sub-) árvore é vazia, deve ser substituída por uma árvore cujo único nó (o nó raiz) contém o valor v . Se a árvore não é vazia, comparamos v com o valor na raiz da árvore, e inserimos v na *sae* ou na *sad*, conforme o resultado da comparação. A função abaixo ilustra a implementação dessa operação. A função tem como valor de retorno o eventual novo nó raiz da (sub-) árvore.

```
Arv* insere (Arv* a, int v)
{
    if (a==NULL) {
        a = (Arv*)malloc(sizeof(Arv));
        a->info = v;
        a->esq = a->dir = NULL;
    }
    else if (v < a->info)
        a->esq = insere(a->esq, v);
    else /* v < a->info */
        a->dir = insere(a->dir, v);
    return a;
}
```

Operação de remoção

Outra operação a ser analisada é a que permite retirar um determinado elemento da árvore. Essa operação é um pouco mais complexa que a de inserção. Existem três situações possíveis. A primeira, e mais simples, é quando se deseja retirar um elemento que é folha da árvore (isto é, um elemento que não tem filhos). Neste caso, basta retirar o elemento da árvore e atualizar o pai, pois seu filho não existe mais.

A segunda situação, ainda simples, acontece quando o nó a ser retirado possui um único filho. Para retirar esse elemento é necessário antes acertar o ponteiro do pai, “pulando” o nó: o único neto passa a ser filho direto. A Figura 16.2 ilustra esse procedimento.

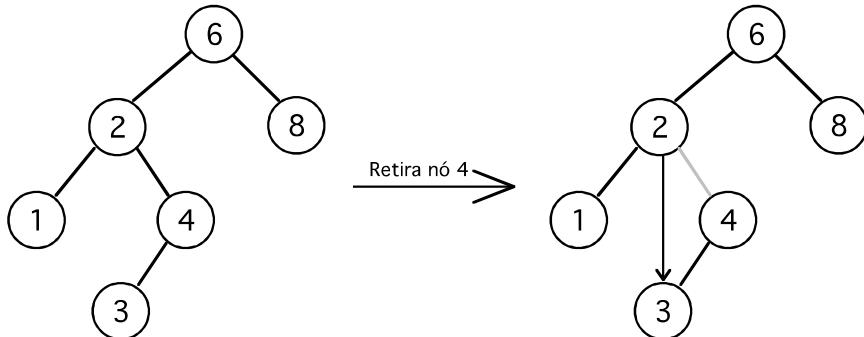


Figura 16.2: Retirada de um elemento com um único filho.

O caso complicado ocorre quando o nó a ser retirado tem dois filhos. Para poder retirar esse nó da árvore, devemos proceder da seguinte forma:

- encontramos o elemento que precede o elemento a ser retirado na ordenação. Isto equivale a encontrar o elemento mais à direita da sub-árvore à esquerda;
- trocamos a informação do nó a ser retirado com a informação do nó encontrado;
- retiramos o nó encontrado (que agora contém a informação do nó que se deseja retirar). Observa-se que retirar o nó mais à direita é trivial, pois esse é um nó folha ou um nó com um único filho (no caso, o filho da direita nunca existe).

O procedimento descrito acima deve ser seguido para não haver violação da ordenação da árvore. Observamos que, análogo ao que foi feito com o nó mais à direita da sub-árvore à esquerda, pode ser feito com o nó mais à esquerda da sub-árvore à direita (que é o nó que segue o nó a ser retirado na ordenação).

A Figura 16.3 exemplifica a retirada de um nó com dois filhos. Na figura é mostrada a estratégia de retirar o elemento que precede o elemento a ser retirado na ordenação.

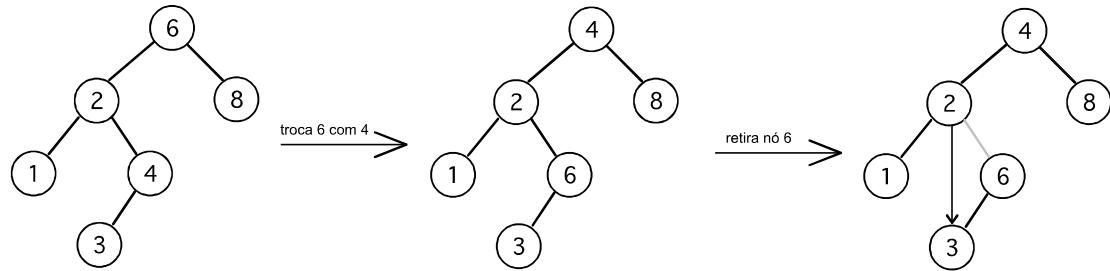


Figura 16.3: Exemplo da operação para retirar o elemento com informação igual a 6.

O código abaixo ilustra a implementação da função para retirar um elemento da árvore binária de busca. A função tem como valor de retorno a eventual nova raiz da (sub-)árvore.

```

Arv* retira (Arv* r, int v)
{
    if (r == NULL)
        return NULL;
    else if (r->info > v)
        r->esq = retira(r->esq, v);
    else if (r->info < v)
        r->dir = retira(r->dir, v);
    else { /* achou o elemento */
        if (r->esq == NULL && r->dir == NULL) { /* elemento sem filhos */
            free (r);
            r = NULL;
        }
        else if (r->esq == NULL) { /* só tem filho à direita */
            Arv* t = r;
            r = r->dir;
            free (t);
        }
        else if (r->dir == NULL) { /* só tem filho à esquerda */
            Arv* t = r;
            r = r->esq;
            free (t);
        }
        else { /* tem os dois filhos */
            Arv* pai = r;
            Arv* f = r->esq;
            while (f->dir != NULL) {
                pai = f;
                f = f->dir;
            }
            r->info = f->info; /* troca as informações */
            f->info = v;
            r->esq = retira(r->esq, v);
        }
    }
    return r;
}

```

Exercício: Escreva um programa que utilize as funções de árvore binária de busca mostradas acima.

Árvores balanceadas

É fácil prever que, após várias operações de inserção/remoção, a árvore tende a ficar desbalanceada, já que essas operações, conforme descritas, não garantem o balanceamento. Em especial, nota-se que a função de remoção favorece uma das sub-árvore (sempre retirando um nó da sub-árvore à esquerda, por exemplo). Uma estratégia que pode ser utilizada para amenizar o problema é intercalar de qual sub-árvore será retirado o nó. No entanto, isso ainda não garante o balanceamento da árvore.

Para que seja possível usar árvores binárias de busca mantendo sempre a altura das árvores no mínimo, ou próximo dele, é necessário um processo de inserção e remoção de nós mais complicado, que mantenha as árvores “balanceadas”, ou “equilibradas”, tendo as duas sub-árvore de cada nó o mesmo “peso”, ou pesos aproximadamente iguais. No caso de um número de nós par, podemos aceitar uma diferença de um nó entre a *sae* (sub-árvore à esquerda) e a *sad* (sub-árvore à direita).

A idéia central de um algoritmo para balancear (equilibrar) uma árvore binária de busca pode ser a seguinte: se tivermos uma árvore com m elementos na *sae*, e $n \geq m + 2$ elementos na *sad*, podemos tornar a árvore menos desequilibrada movendo o valor da raiz para a *sae*, onde ele se tornará o maior valor, e movendo o menor elemento da *sad* para a raiz. Dessa forma, a árvore continua com os mesmos elementos na mesma ordem. A situação em que a *sad* tem menos elementos que a *sae* é semelhante. Esse processo pode ser repetido até que a diferença entre os números de elementos das duas sub-árvore seja menor ou igual a 1. Naturalmente, o processo deve continuar (recursivamente) com o balanceamento das duas sub-árvore de cada árvore. Um ponto a observar é que remoção do menor (ou maior) elemento de uma árvore é mais simples do que a remoção de um elemento qualquer.

Exercício: Implemente o algoritmo para balanceamento de árvore binária descrito acima.

17. Tabelas de dispersão

W. Celes e J. L. Rangel

No capítulo anterior, discutimos diferentes estruturas e algoritmos para buscar um determinado elemento num conjunto de dados. Para obtermos algoritmos eficientes, armazenamos os elementos ordenados e tiramos proveito dessa ordenação para alcançar eficientemente o elemento procurado. Chegamos a conclusão que os algoritmos eficientes de busca demandam um esforço computacional de $O(\log n)$. Neste capítulo, vamos estudar as estruturas de dados conhecidas como tabelas de dispersão (*hash tables*), que, se bem projetadas, podem ser usadas para buscar um elemento da tabela em ordem constante: $O(1)$. O preço pago por essa eficiência será um uso maior de memória, mas, como veremos, esse uso excedente não precisa ser tão grande, e é proporcional ao número de elementos armazenados.

Para apresentar a idéia das tabelas de dispersão, vamos considerar um exemplo onde desejamos armazenar os dados referentes aos alunos de uma disciplina. Cada aluno é individualmente identificado pelo seu número de matrícula. Podemos então usar o número de matrícula como chave de busca do conjunto de alunos armazenados. Na PUC-Rio, o número de matrícula dos alunos é dado por uma seqüência de oito dígitos, sendo que o último representa um dígito de controle, não sendo portanto parte efetiva do número de matrícula. Por exemplo, na matrícula 9711234-4, o ultimo dígito 4, após o hífen, representa o dígito de controle. O número de matrícula efetivo nesse caso é composto pelos primeiros sete dígitos: 9711234.

Para permitir um acesso a qualquer aluno em ordem constante, podemos usar o número de matrícula do aluno como índice de um vetor – `vet`. Se isso for possível, acessamos os dados do aluno cuja matrícula é dado por `mat` indexando o vetor – `vet[mat]`. Dessa forma, o acesso ao elemento se dá em ordem constante, imediata. O problema que encontramos é que, nesse caso, o preço pago para se ter esse acesso rápido é muito grande.

Vamos considerar que a informação associada a cada aluno seja representada pela estrutura abaixo:

```
struct aluno {
    int mat;
    char nome[81];
    char email[41];
    char turma;
};
typedef struct aluno Aluno;
```

Como a matrícula é composta por sete dígitos, o número inteiro que conceitualmente representa uma matrícula varia de 0000000 a 9999999. Portanto, precisamos dimensionar nosso vetor com dez milhões (10.000.000) de elementos. Isso pode ser feito por:

```
#define MAX 10000000
Aluno vet[MAX];
```

Dessa forma, o nome do aluno com matrícula `mat` é acessado simplesmente por: `vet[mat].nome`. Temos um acesso rápido, mas pagamos um preço em uso de

memória proibitivo. Como a estrutura de cada aluno, no nosso exemplo, ocupa pelo menos 127 bytes, estamos falando num gasto de 1.270.000.000 bytes, ou seja, acima de 1 Gbyte de memória. Como na prática teremos, digamos, em torno de 50 alunos cadastrados, precisaríamos apenas de algo em torno de 6.350 ($=127*50$) bytes.

Para amenizar o problema, já vimos que podemos ter um vetor de ponteiros, em vez de um vetor de estruturas. Desta forma, as posições do vetor que não correspondem a alunos cadastrados teriam valores NULL. Para cada aluno cadastrado, alocaríamos dinamicamente a estrutura de aluno e armazenaríamos um ponteiro para essa estrutura no vetor. Neste caso, acessaríamos o nome do aluno de matrícula `mat` por `vet[mat] ->nome`. Assim, considerando que cada ponteiro ocupe 4 bytes, o gasto excedente de memória seria, no máximo, aproximadamente 40 Mbytes. Apesar de menor, esse gasto de memória ainda é proibitivo.

A forma de resolver o problema de gasto excessivo de memória, mas ainda garantindo um acesso rápido, é através do uso de tabelas de dispersão (*hash table*) que discutiremos a seguir.

17.1. Idéia central

A idéia central por trás de uma tabela de dispersão é identificar, na chave de busca, quais as partes significativas. Na PUC-Rio, por exemplo, além do dígito de controle, alguns outros dígitos do número de matrícula têm significados especiais, conforme ilustra a Figura 17.1.

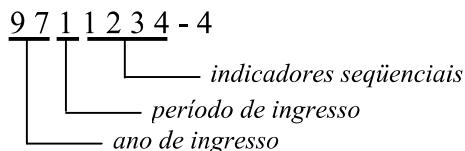


Figura 17.1: Significado dos dígitos do número da matrícula.

Numa turma de aluno, é comum existirem vários alunos com o mesmo ano e período de ingresso. Portanto, esses três primeiros dígitos não são bons candidatos para identificar individualmente cada aluno. Reduzimos nosso problema para uma chave com os quatro dígitos seqüenciais. Podemos ir além e constatar que os números seqüenciais mais significativos são os últimos, pois num universo de uma turma de alunos, o dígito que representa a unidade varia mais do que o dígito que representa o milhar.

Desta forma, podemos usar um número de matrícula parcial, de acordo com a dimensão que queremos que tenha nossa tabela (ou nosso vetor). Por exemplo, para dimensionarmos nossa tabela com apenas 100 elementos, podemos usar apenas os últimos dois dígitos seqüenciais do número de matrícula. A tabela pode então ser declarada por: `Aluno* tab[100]`.

Para acessarmos o nome do aluno cujo número de matrícula é dado por `mat`, usamos como índice da tabela apenas os dois últimos dígitos. Isso pode ser conseguido aplicando-se o operador módulo (%): `vet[mat%100] ->nome`.

Desta forma, o uso de memória excedente é pequeno e o acesso a um determinado aluno, a partir do número de matrícula, continua imediato. O problema que surge é que provavelmente existirão dois ou mais alunos da turma que apresentarão os mesmos últimos dois dígitos no número de matrícula. Dizemos que há uma *colisão*, pois alunos diferentes são mapeados para o mesmo índice da tabela. Para que a estrutura funcione de maneira adequada, temos que resolver esse problema, tratando as colisões.

Existem diferentes métodos para tratarmos as colisões em tabelas de dispersão, e estudaremos esses métodos mais adiante. No momento, vale salientar que não há como eliminar completamente a ocorrência de colisões em tabelas de dispersão. Devemos minimizar as colisões e usar um método que, mesmo com colisões, saibamos identificar cada elemento da tabela individualmente.

17.2. Função de dispersão

A função de dispersão (função de *hash*) mapeia uma chave de busca num índice da tabela. Por exemplo, no caso exemplificado acima, adotamos como função de *hash* a utilização dos dois últimos dígitos do número de matrícula. A implementação dessa função recebe como parâmetro de entrada a chave de busca e retorna um índice da tabela. No caso da chave de busca ser um inteiro representando o número de matrícula, essa função pode ser dada por.

```
int hash (int mat)
{
    return (mat%100);
}
```

Podemos generalizar essa função para tabelas de dispersão com dimensão N . Basta avaliar o módulo do número de matrícula por N :

```
int hash (int mat)
{
    return (mat%N);
}
```

Uma função de *hash* deve, sempre que possível, apresentar as seguintes propriedades:

Ser eficientemente avaliada: isto é necessário para termos acesso rápido, pois temos que avaliar a função de *hash* para determinarmos a posição onde o elemento se encontra armazenado na tabela.

Espalhar bem as chaves de busca: isto é necessário para minimizarmos as ocorrências de colisões. Como veremos, o tratamento de colisões requer um procedimento adicional para encontrarmos o elemento. Se a função de *hash* resulta em muitas colisões, perdemos o acesso rápido aos elementos. Um exemplo de função de *hash* ruim seria usar, como índice da tabela, os dois dígitos iniciais do número de matrícula – todos os alunos iriam ser mapeados para apenas três ou quatro índices da tabela.

Ainda para minimizarmos o número de colisões, a dimensão da tabela deve guardar uma folga em relação ao número de elementos efetivamente armazenados. Como regra empírica, não devemos permitir que a tabela tenha uma taxa de ocupação superior a

75%. Uma taxa de 50% em geral traz bons resultados. Uma taxa menor que 25% pode representar um gasto excessivo de memória.

17.3. Tratamento de colisão

Existem diversas estratégias para tratarmos as eventuais colisões que surgem quando duas ou mais chaves de busca são mapeadas para um mesmo índice da tabela de *hash*. Nesta seção, vamos apresentar algumas dessas estratégias comumente usadas. Para cada uma das estratégias, vamos apresentar as duas principais funções de manipulação de tabelas de dispersão: a função que busca um elemento na tabela e a função que insere ou modifica um elemento. Nessas implementações, vamos considerar a existência da função de dispersão que mapeia o número de matrícula num índice da tabela, vista na seção anterior.

Em todas as estratégias, a tabela de dispersão em si é representada por um vetor de ponteiros para a estrutura que representa a informação a ser armazenada, no caso Aluno. Podemos definir um tipo que representa a tabela por:

```
#define N 100
typedef Aluno* Hash[N];
```

Uso da posição consecutiva livre

Nas duas primeiras estratégias que discutiremos, os elementos que colidem são armazenados em outros índices, ainda não ocupados, da própria tabela. A escolha da posição ainda não ocupada para armazenar um elemento que colide diferencia as estratégias que iremos discutir. Numa primeira estratégia, se a função de dispersão mapeia para um índice já ocupado, procuramos o próximo (usando incremento circular) índice livre da tabela para armazenar o novo elemento. A Figura 17.2 ilustra essa estratégia. Nessa figura, os índices da tabela que não têm elementos associados são preenchidos com o valor NULL.

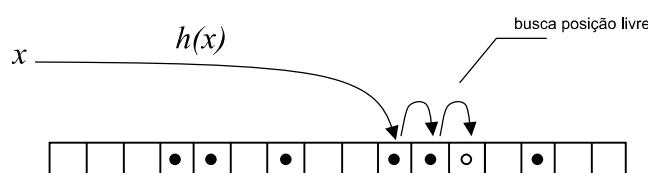


Figura 17.2: Tratamento de colisões usando próxima posição livre.

Vale lembrar que uma tabela de dispersão nunca terá todos os elementos preenchidos (já mencionamos que uma ocupação acima de 75% eleva o número de colisões, descaracterizando a idéia central da estrutura). Portanto, podemos garantir que sempre existirá uma posição livre na tabela.

Na operação de busca, considerando a existência de uma tabela já construída, se uma chave x for mapeada pela função de dispersão (função de *hash* – h) para um determinado índice $h(x)$, procuramos a ocorrência do elemento a partir desse índice, até que o elemento seja encontrado ou que uma posição vazia seja encontrada. Uma possível implementação é mostrada a seguir. Essa função de busca recebe, além da

tabela, a chave de busca do elemento que se busca, e tem como valor de retorno o ponteiro do elemento, se encontrado, ou NULL no caso do elemento não estar presente na tabela.

```
Aluno* busca (Hash tab, int mat)
{
    int h = hash(mat);
    while (tab[h] != NULL) {
        if (tab[h]->mat == mat)
            return tab[h];
        h = (h+1) % N;
    }
    return NULL;
}
```

Devemos notar que a existência de algum elemento mapeado para o mesmo índice não garante que o elemento que buscamos esteja presente. A partir do índice mapeado, temos que buscar o elemento utilizando, como chave de comparação, a real chave de busca, isto é, o número de matrícula completo.

A função que insere ou modifica um determinado elemento também é simples. Fazemos o mapeamento da chave de busca (no caso, número de matrícula) através da função de dispersão e verificamos se o elemento já existe na tabela. Se o elemento existir, modificamos o seu conteúdo; se não existir, inserimos um novo na primeira posição que encontrarmos na tabela, a partir do índice mapeado. Uma possível implementação dessa função é mostrada a seguir. Essa função recebe como parâmetros a tabela e os dados do elemento sendo inserido (ou os novos dados de um elemento já existente). A função tem como valor de retorno o ponteiro do aluno modificado ou do novo aluno inserido.

```
Aluno* insere (Hash tab, int mat, char* nome, char* email, char turma)
{
    int h = hash(mat);
    while (tab[h] != NULL) {
        if (tab[h]->mat == mat)
            break;
        h = (h+1) % N;
    }
    if (tab[h]==NULL) {           /* não encontrou o elemento */
        tab[h] = (Aluno*) malloc(sizeof(Aluno));
        tab[h]->mat = mat;
    }
    /* atribui informação */
    strcpy(tab[h]->nome,nome);
    strcpy(tab[h]->email,email);
    tab[h]->turma = turma;
    return tab[h];
}
```

Apesar de bastante simples, essa estratégia tende a concentrar os lugares ocupados na tabela, enquanto que o ideal seria dispersar. Uma estratégia que visa melhorar essa concentração é conhecida como “dispersão dupla” (*double hash*) e será apresentada a seguir.

Uso de uma segunda função de dispersão

Para evitar a concentração de posições ocupadas na tabela, essa segunda estratégia faz uma variação na forma de procurar uma posição livre a fim armazenar o elemento que

colidiu. Aqui, usamos uma segunda função de dispersão, h' . Para chaves de busca dadas por números inteiros, uma possível segunda função de dispersão é definida por:

$$h'(x) = N - 2 - x\%(N - 2)$$

Nesta fórmula, x representa a chave de busca e N a dimensão da tabela. De posse dessa segunda função, procuramos uma posição livre na tabela com incrementos, ainda circulares, dados por $h'(x)$. Isto é, em vez de tentarmos $(h(x)+1)\%N$, tentamos $(h(x)+h'(x))\%N$. Dois cuidados devem ser tomados na escolha dessa segunda função de dispersão: primeiro, ela nunca pode retornar zero, pois isso não varia com que o índice fosse incrementado; segundo, de preferência, ela não pode retornar um número divisor da dimensão da tabela, pois isso nos limitaria a procurar uma posição livre num subconjunto restrito dos índices da tabela.

A implementação da função de busca com essa estratégia é uma pequena variação da função de busca apresentada para a estratégia anterior.

```
int hash2 (int mat)
{
    return N - 2 - mat%(N-2);
}

Aluno* busca (Hash tab, int mat)
{
    int h = hash(mat);
    int h2 = hash2(mat);
    while (tab[h] != NULL) {
        if (tab[h]->mat == mat)
            return tab[h];
        h = (h+h2) % N;
    }
    return NULL;
}
```

Exercício: Implemente a função para inserir (ou modificar) um elemento usando a estratégia de uma segunda função de dispersão.

Uso de listas encadeadas

Uma estratégia diferente, mas ainda simples, consiste em fazer com que cada elemento da tabela *hash* represente um ponteiro para uma lista encadeada. Todos os elementos mapeados para um mesmo índice seriam armazenados na lista encadeada. A Figura 17.1 ilustra essa estratégia. Nessa figura, os índices da tabela que não têm elementos associados representam listas vazias.

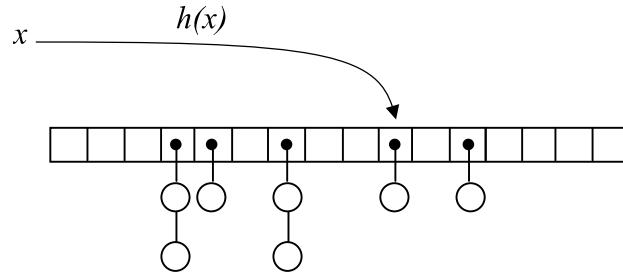


Figura 17.3: Tratamento de colisões com lista encadeada.

Com essa estratégia, cada elemento armazenado na tabela será um elemento de uma lista encadeada. Portanto, devemos prever, na estrutura da informação, um ponteiro adicional para o próximo elemento da lista. Nossa estrutura de aluno passa a ser dada por:

```
struct aluno {
    int mat;
    char nome[81];
    char turma;
    char email[41];
    struct aluno* prox; /* encadeamento na lista de colisão */
};

typedef struct aluno Aluno;
```

Na operação de busca, procuramos a ocorrência do elemento na lista representada no índice mapeado pela função de dispersão. Uma possível implementação é mostrada a seguir.

```
Aluno* busca (Hash tab, int mat)
{
    int h = hash(mat);
    Aluno* a = tab[h];
    while (a != NULL) {
        if (a->mat == mat)
            return a;
        a = a->prox;
    }
    return NULL;
}
```

A função que insere ou modifica um determinado elemento também é simples e pode ser dada por:

```

Aluno* insere (Hash tab, int mat, char* nome, char turma)
{
    int h = hash(mat);
    Aluno* p = NULL;           /* ponteiro para anterior */
    Aluno* a = tab[h];
    while (a != NULL) {
        if (a->mat == mat)
            break;
        p = a;
        a = a->prox;
    }
    if (a==NULL) {             /* não encontrou o elemento */
        a = (Aluno*) malloc(sizeof(Aluno));
        a->mat = mat;
        a->prox = NULL;
        if (p==NULL)
            tab[h] = a;
        else
            p->prox = a;
    }
    /* atribui informação */
    strcpy(a->nome,nome);
    a->turma = turma;
    return a;
}

```

Exercício: Faça um programa que utilize as funções de tabelas de dispersão vistas acima.

17.4. Exemplo: Número de Ocorrências de Palavras

Para exemplificar o uso de tabelas de dispersão, vamos considerar o desenvolvimento de um programa para exibir quantas vezes cada palavra foi utilizada em um dado texto. A saída do programa será uma lista de palavras, em ordem decrescente do número de vezes que cada palavra ocorre no texto de entrada. Para simplificar, não consideraremos caracteres acentuados.

Projeto: “Dividir para conquistar”

A melhor estratégia para desenvolvermos programas é dividirmos um problema grande em diversos problemas menores. Uma aplicação deve ser construída através de módulos independentes. Cada módulo é projetado para a realização de tarefas específicas. Um segundo módulo, que é cliente, não precisa conhecer detalhes de como o primeiro foi implementado; o cliente precisa apenas saber a funcionalidade oferecida pelo módulo que oferece os serviços. Dentro de cada módulo, a realização da tarefa é dividida entre várias pequenas funções. Mais uma vez, vale a mesma regra de encapsulamento: funções clientes não precisam conhecer detalhes de implementação das funções que oferecem os serviços. Dessa forma, aumentamos o potencial de re-uso do código e facilitamos o entendimento e a manutenção do programa.

O programa para contar o uso das palavras é um programa relativamente simples, que não precisa ser subdividido em módulos para ser construído. Aqui, vamos projetar o programa identificando as diversas funções necessárias para a construção do programa como um todo. Cada função tem sua finalidade específica e o programa principal (a função `main`) fará uso dessas funções.

Vamos considerar que uma palavra se caracteriza por uma seqüência de uma ou mais letras (maiúsculas ou minúsculas). Para contar o número de ocorrências de cada palavra, podemos armazenar as palavras lidas numa tabela de dispersão, usando a própria palavra como chave de busca. Guardaremos na estrutura de dados quantas vezes cada palavra foi encontrada. Para isso, podemos prever a construção de uma função que acessa uma palavra armazenada na tabela; se a palavra ainda não existir, a função armazena uma nova palavra na tabela. Dessa forma, para cada palavra lida, conseguiremos incrementar o número de ocorrências. Para exibir as ocorrências em ordem decrescente, criaremos um vetor e armazenaremos todas as palavras que existem na tabela de dispersão no vetor. Esse vetor pode então ser ordenado e seu conteúdo exibido.

Tipo dos dados

Conforme discutido acima, usaremos uma tabela de dispersão para contar o número de ocorrências de cada palavra no texto. Vamos optar por empregar a estratégia que usa lista encadeada para o tratamento de colisões. Dessa forma, a dimensão da tabela de dispersão não compromete o número máximo de palavras distintas (no entanto, a dimensão da tabela não pode ser muito justa em relação ao número de elementos armazenados, pois aumentaria o número de colisões, degradando o desempenho). A estrutura que define a tabela de dispersão pode ser dada por:

```
#define NPAL 64 /* dimensão máxima de cada palavra */
#define NTAB 127 /* dimensão da tabela de dispersão */

/* tipo que representa cada palavra */
struct palavra {
    char pal[NPAL];
    int n;
    struct palavra* prox; /* tratamento de colisão com listas */
};
typedef struct palavra Palavra;

/* tipo que representa a tabela de dispersão */
typedef Palavra* Hash[NTAB];
```

Leitura de palavras

A primeira função que vamos discutir é responsável por capturar a próxima seqüência de letras do arquivo texto. Essa função receberá como parâmetros o ponteiro para o arquivo de entrada e a cadeia de caracteres que armazenará a palavra capturada. A função tem como valor de retorno um inteiro que indica se a leitura foi bem sucedida (1) ou não (0). A próxima palavra é capturada pulando os caracteres que não são letras e, então, capturando a seqüência de letras do arquivo. Para identificar se um caractere é ou não letra, usaremos a função `isalpha` disponibilizada pela interface `ctype.h`.

```

int le_palavra (FILE* fp, char* s)
{
    int i = 0;
    int c;

    /* pula caracteres que não são letras */
    while ((c = fgetc(fp)) != EOF) {
        if (isalpha(c))
            break;
    };

    if (c == EOF)
        return 0;
    else
        s[i++] = c;      /* primeira letra já foi capturada */

    /* lê os próximos caracteres que são letras */
    while (i < NPAL-1 && (c = fgetc(fp)) != EOF && isalpha(c))
        s[i++] = c;
    s[i] = '\0';

    return 1;
}

```

Tabela de dispersão com cadeia de caracteres

Devemos implementar as funções responsáveis por construir e manipular a tabela de dispersão. A primeira função que precisamos é responsável por inicializar a tabela, atribuindo o valor NULL para cada elemento.

```

void inicializa (Hash tab)
{
    int i;
    for (i=0; i<NTAB; i++)
        tab[i] = NULL;
}

```

Também precisamos definir uma função de dispersão, responsável por mapear a chave de busca, uma cadeia de caracteres, em um índice da tabela. Uma função de dispersão simples para cadeia de caracteres consiste em somar os código dos caracteres que compõem a cadeia e tirar o módulo dessa soma para se obter o índice da tabela. A implementação abaixo ilustra essa função.

```

int hash (char* s)
{
    int i;
    int total = 0;
    for (i=0; s[i]!='\0'; i++)
        total += s[i];
    return total % NTAB;
}

```

Precisamos ainda da função que acessa os elementos armazenados na tabela. Criaremos uma função que, dada uma palavra (chave de busca), fornece como valor de retorno o ponteiro da estrutura Palavra associada. Se a palavra ainda não existir na tabela, essa função cria uma nova palavra e fornece como retorno essa nova palavra criada.

```

Palavra *acessa (Hash tab, char* s)
{
    int h = hash(s);
    Palavra* p;
    for (p=tab[h]; p!=NULL; p=p->prox) {
        if (strcmp(p->pal,s) == 0)
            return p;
    }
    /* insere nova palavra no inicio da lista */
    p = (Palavra*) malloc(sizeof(Palavra));
    strcpy(p->pal,s);
    p->n = 0;
    p->prox = tab[h];
    tab[h] = p;
    return p;
}

```

Dessa forma, a função cliente será responsável por acessar cada palavra e incrementar o seu número de ocorrências. Transcrevemos abaixo o trecho da função principal responsável por fazer essa contagem (a função completa será mostrada mais adiante).

```

...
inicializa(tab);
while (le_palavra(fp,s)) {
    Palavra* p = acessa(tab,s);
    p->n++;
}
...

```

Com a execução desse trecho de código, cada palavra encontrada no texto de entrada será armazenada na tabela, associada ao número de vezes de sua ocorrência. Resta-nos arrumar o resultado obtido para podermos exibir as palavras em ordem decrescente do número de ocorrências.

Exibição do resultado ordenado

Para colocarmos o resultado na ordem desejada, criaremos dinamicamente um vetor para armazenar as palavras. Optaremos por construir um vetor de ponteiros para a estrutura `Palavra`. Esse vetor será então ordenado em ordem decrescente do número de ocorrências de cada palavra; se duas palavras tiverem o mesmo número de ocorrências, usaremos a ordem alfabética como critério de desempate.

Para criar o vetor, precisamos conhecer o número de palavras armazenadas na tabela de dispersão. Podemos implementar uma função que percorre a tabela e conta o número de palavras existentes. Essa função pode ser dada por:

```

int conta_elems (Hash tab)
{
    int i;
    int total = 0;
    Palavra* p;
    for (i=0; i<NTAB; i++) {
        for (p=tab[i]; p!=NULL; p=p->prox)
            total++;
    }
    return total;
}

```

Podemos agora implementar a função que cria dinamicamente o vetor de ponteiros. Em seguida, a função percorre os elementos da tabela e preenche o conteúdo do vetor. Essa

função recebe como parâmetros de entrada o número de elementos e a tabela de dispersão.

```
Palavra** cria_vetor (int n, Hash tab)
{
    int i, j=0;
    Palavra* p;
    Palavra** vet = (Palavra**) malloc(n*sizeof(Palavra*));

    /* percorre tabela preenchendo vetor */
    for (i=0; i<NTAB; i++) {
        for (p=tab[i]; p!=NULL; p=p->prox)
            vet[j++] = p;
    }
    return vet;
}
```

Para ordenar o vetor (de ponteiros para a estrutura Palavra) utilizaremos a função `qsort` da biblioteca padrão. Precisamos então definir a função de comparação, que é mostrada abaixo.

```
int compara (const void* v1, const void* v2)
{
    Palavra** pp1 = (Palavra**)v1;
    Palavra** pp2 = (Palavra**)v2;
    Palavra* p1 = *pp1;
    Palavra* p2 = *pp2;

    if (p1->n > p2->n) return -1;
    else if (p1->n < p2->n) return 1;
    else return strcmp(p1->pal,p2->pal);
}
```

Por fim, podemos escrever a função que, dada a tabela de dispersão já preenchida e utilizando as funções mostradas acima, conta o número de elementos, cria o vetor, ordena-o e exibe o resultado na ordem desejada. Ao final, a função libera o vetor criado dinamicamente.

```
void imprime (Hash tab)
{
    int i;
    int n;
    Palavra** vet;

    /* cria e ordena vetor */
    n = conta_elems(tab);
    vet = cria_vetor(n,tab);
    qsort(vet,n,sizeof(Palavra*),compara);

    /* imprime ocorrencias */
    for (i=0; i<n; i++)
        printf("%s = %d\n",vet[i]->pal,vet[i]->n);

    /* libera vetor */
    free(vet);
}
```

Função principal

Uma possível função principal desse programa é mostrada a seguir. Esse programa espera receber como dado de entrada o nome do arquivo cujas palavras queremos contar o número de ocorrências. Para exemplificar a utilização dos parâmetros da função

principal, utilizamos esses parâmetros para receber o nome do arquivo de entrada (para detalhes, veja seção 6.3).

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

... /* funções auxiliares mostradas acima */

int main (int argc, char** argv)
{
    FILE* fp;
    Hash tab;
    char s[NPAL];

    if (argc != 2) {
        printf("Arquivo de entrada nao fornecido.\n");
        return 0;
    }

    /* abre arquivo para leitura */
    fp = fopen(argv[1],"rt");
    if (fp == NULL) {
        printf("Erro na abertura do arquivo.\n");
        return 0;
    }

    /* conta ocorrencia das palavras */
    inicializa(tab);
    while (le_palavra(fp,s)) {
        Palavra* p = acessa(tab,s);
        p->n++;
    }

    /* imprime ordenado */
    imprime (tab);

    return 0;
}
```

17.5. Uso de callbacks **

No programa da seção anterior, precisamos implementar duas funções que percorrem os elementos da tabela de dispersão: uma para contar o número de elementos e outra para preencher o vetor. Em todas as estruturas de dados, é muito comum necessitarmos de funções que percorrem os elementos, executando uma ação específica para cada elemento. Como um exemplo adicional, podemos imaginar uma função para imprimir os elementos na ordem em que eles aparecem na tabela de dispersão. Ainda usando o exemplo da seção anterior, poderíamos ter:

```
void imprime_tabela (Hash tab)
{
    int i;
    Palavra* p;

    for (i=0; i<NTAB; i++) {
        for (p=tab[i]; p!=NULL; p=p->prox)
            printf("%s = %d\n",p->pal,p->n);
    }
}
```

Podemos observar que as estruturas dessas funções são as mesmas. Como também teriam as mesmas estruturas funções para percorrer os elementos de uma lista encadeada, de uma árvore, etc.

Nesses casos, podemos separar a função que percorre os elementos da ação que realizamos a cada elemento. Assim, a função que percorre os elementos é única e pode ser usada para diversos fins. A ação que deve ser executada é passada como parâmetro, via um ponteiro para uma função. Essa função é usualmente chamada de *callback* pois é uma função do cliente (quem usa a função que percorre os elementos) que é “chamada de volta” a cada elemento encontrado na estrutura de dados. Usualmente essa função *callback* recebe como parâmetro o elemento encontrado na estrutura. No nosso exemplo, como os elementos são ponteiros para a estrutura Palavra, a função receberia o ponteiro para cada palavra encontrada na tabela.

Uma função genérica para percorrer os elementos da tabela de dispersão do nosso exemplo pode ser dada por:

```
void percorre (Hash tab, void (*cb) (Palavra*) )
{
    int i;
    Palavra* p;

    for (i=0; i<NTAB; i++) {
        for (p=tab[i]; p!=NULL; p=p->prox)
            cb(p);
    }
}
```

Para ilustrar sua utilização, podemos usar essa função para imprimir os elementos. Para tanto, devemos escrever a função que executa a ação de imprimir cada elemento. Essa função pode ser dada por:

```
void imprime_elemento (Palavra* p)
{
    printf("%s = %d\n", p->pal, p->n);
}
```

Assim, para imprimir os elementos da tabela bastaria chamar a função *percorre* com a ação acima passada como parâmetro.

```
...
percorre(tab, imprime_elemento);
...
```

Essa mesma função *percorre* pode ser usada para, por exemplo, contar o número de elementos que existe armazenado na tabela. A ação associada aqui precisa apenas incrementar um contador do número de vezes que a *callback* é chamada. Para tanto, devemos usar uma variável global que representa esse contador e fazer a *callback* incrementar esse contador cada vez que for chamada. Nesse caso, o ponteiro do elemento passado como parâmetro para a *callback* não é utilizado, pois o incremento ao contador independe do elemento. Assumindo que *Total* é uma variável global inicializada com o valor zero, a ação para contar o número de elementos é dada simplesmente por:

```

void conta_elemento (Palavra* p)
{
    Total++;
}

```

Já mencionamos que o uso de variáveis globais deve, sempre que possível, ser evitado, pois seu uso indiscriminado torna um programa ilegível e difícil de ser mantido. Para evitar o uso de variáveis globais nessas funções *callbacks* devemos arrumar um meio de transferir, para a função *callback*, um dado do cliente. A função que percorre os elementos não manipula esse dado, apenas o transfere para a função *callback*. Como não sabemos *a priori* o tipo de dado que será necessário, definimos a *callback* recebendo dois parâmetros: o elemento sendo visitado e um ponteiro genérico (`void*`). O cliente chama a função que percorre os elementos passando como parâmetros a função *callback* e um ponteiro a ser repassado para essa mesma *callback*.

Vamos exemplificar o uso dessa estratégia re-implementando a função que percorre os elementos.

```

void percorre (Hash tab, void (*cb) (Palavra*, void*), void* dado)
{
    int i;
    Palavra* p;

    for (i=0; i<NTAB; i++) {
        for (p=tab[i]; p!=NULL; p=p->prox)
            cb(p, dado); /* passa para a callback o ponteiro recebido */
    }
}

```

Agora, podemos usar essa nova versão da função para contar o número de elementos, sem usar variável global. Primeiro temos que definir a *callback*, que, nesse caso, receberá um ponteiro para um inteiro que representa o contador.

```

void conta_elemento (Palavra* p, void* dado)
{
    int *contador = (int*)dado;
    (*contador)++;
}

```

Por fim, uma função que conta o número de elemento, usando as funções acima, é mostrada a seguir.

```

int total_elementos (Hash tab)
{
    int total = 0;
    percorre(tab, conta_elemento, &total);
    return total;
}

```