MC102 - Algoritmos e Progração de Computador

Prof. Alexandre Xavier Falcão

3 de março de 2005

Sumário

Capítulo 1

Introdução

1.1 Organização do computador

O computador é uma máquina que funciona mediante instruções enviadas pelo ser humano ou por outra máquina, executando tarefas e resolvendo problemas tais como: cálculos complexos, geração de relatórios, comando de outras máquinas (e.g. robô), controle de contas bancárias, comunicação de informações, etc. Uma visão simplificada do computador é ilustrada na Figura 1.1.

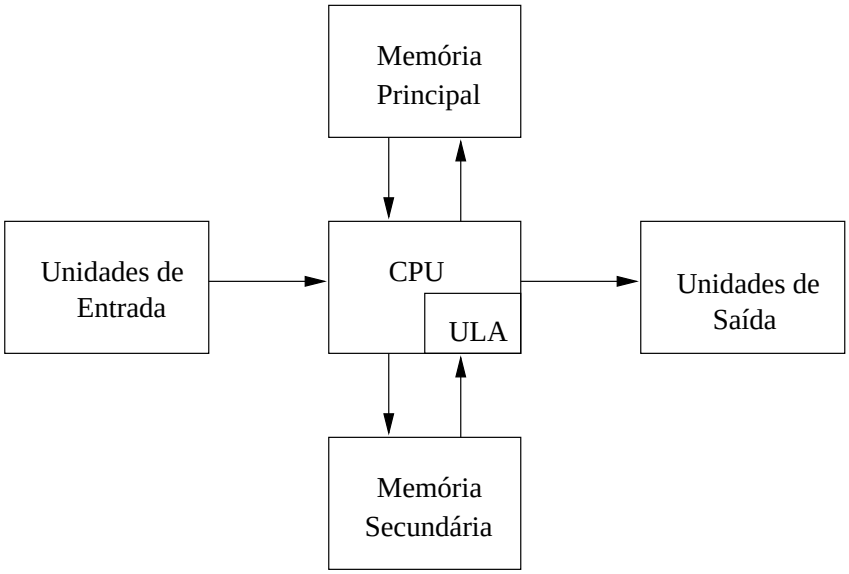

Figura 1.1: Organização básica de um computador

- **Unidades de Entrada:** Usadas pelo computador para receber instruções ou informações externas. Ex: teclado, *mouse*, câmera de vídeo, etc.
- **Unidades de Saída:** Usadas pelo computador para exibir os resultados da computação. Ex: monitor, impressora, etc.
- **Unidade Central de Processamento (*Central Processing Unit* - CPU):** Responsável pelo gerenciamento do sistema como um todo, incluindo as memórias e as unidades de entrada e

saída.

- **Unidade Lógica e Aritmética (ULA):** Responsável pelos cálculos matemáticos. Alguns computadores têm esta unidade separada da CPU. Também chamada de có-processador matemático.
- **Memória Principal:** Usada pela CPU para armazenar instruções e informações enquanto o computador está ligado. Também conhecida como memória RAM (*Random Access Memory*).
- **Memória Secundária:** Usada pelo computador para armazenar instruções e informações por prazo indeterminado, independente do estado do computador (ligado ou desligado). Em geral com capacidade de armazenamento bem maior do que a memória RAM, mas de acesso mais lento. Ex: discos rígidos (*winchester*), disquetes, fitas magnéticas, etc.

Observação: As memórias principal e secundária podem ser vistas como unidades de entrada e saída.

1.2 Alguns termos técnicos

1. **Dados:** Qualquer tipo de informação ou instrução que pode ser manipulada pelo computador. Ex: textos, imagens, etc.
2. **Bit:** Unidade básica para armazenamento, processamento e comunicação de dados.
3. **Byte:** Um conjunto de 8 bits.
4. **Palavra (*word*):** Um conjunto de n bytes (e.g. $n = 1, 2, 4, 8$). Os dados são organizados em palavras de n bytes. Os processadores mais modernos processam os dados em palavras de 16 bytes ou 128bits.
5. **Comandos:** São as instruções que fazem com que o computador execute tarefas.
6. **Programa:** É uma seqüência de instruções com alguma finalidade.
7. **Arquivo:** Conjunto de bytes que contém dados. Estes dados podem ser um programa, uma imagem, uma lista de nomes de pessoas, etc.
8. **Software:** É um conjunto de programas com um propósito global em comum.
9. **Hardware:** Consiste da parte física do computador.
10. **Sistema Operacional:** Conjunto de programas que gerenciam e alocam recursos de hardware e software. Ex: Unix, Windows98, OS2, MSDOS, etc.
11. **Linguagem de Programação:** Consiste da sintaxe (gramática) e semântica (significado) utilizada para escrever (ou codificar) um programa.
    (a) **Alto Nível:** Linguagem de codificação de programa independente do tipo de máquina e de fácil utilização pelo ser humano. Ex: Pascal, C, Algol, Cobol, Fortran (1° linguagem em meados de 1950), BASIC, Java, Python, Tcl/Tk, etc.
    (b) **Baixo Nível:** Linguagem de codificação baseada em mnemônicos. Dependente do tipo de máquina e de fácil tradução para a máquina. Conhecida como linguagem assembly.

12. **Linguagem de Máquina:** Conjunto de códigos binários que são compreendidos pela CPU de um dado computador. Dependente do tipo de máquina.

13. **Compilador:** Traduz programas codificados em linguagem de alto ou baixo nível (i.e. código fonte) para linguagem de máquina (i.e. código executável). Ex: O assembler transforma um programa em assembly para linguagem de máquina. Uma vez compilado, o programa pode ser executado em qualquer máquina com o mesmo sistema operacional para o qual o programa foi compilado.

14. **Interpretador:** Traduz o código fonte para código de máquina diretamente em tempo de execução. Exemplos de linguagens interpretadas são BASIC, Python, Tcl/Tk e LISP.

15. **Algoritmos:** São procedimentos ou instruções escritos em linguagem humana antes de serem codificados usando uma linguagem de programação. Uma receita de bolo é um bom exemplo da organização de um algoritmo.

1.3 Objetivos do curso

A Figura 1.2 mostra um diagrama das tarefas básicas para a solução de problemas usando um computador. O objetivo principal deste curso é exercitar estas tarefas definindo vários conceitos de computação e usando a linguagem C como ferramenta de programação.

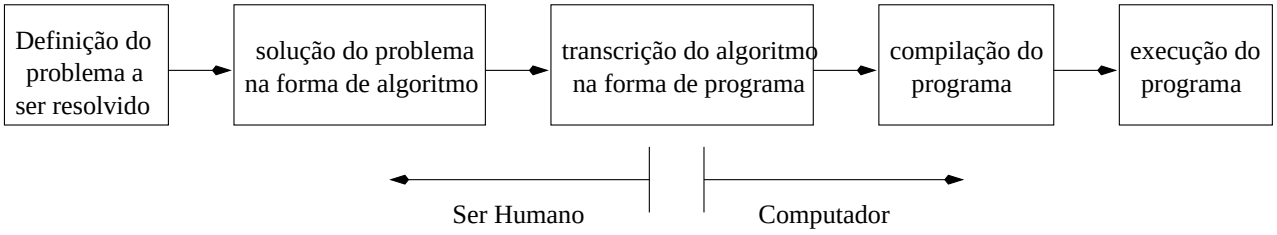

Figura 1.2: Etapas da resolução de problemas usando um computador.

1.4 Jogo das gavetas

Essencialmente, programar um computador para executar uma dada tarefa é estabelecer regras de manipulação de informações na sua memória principal através de uma seqüência de comandos. A memória principal funciona como um armário de gavetas, cuja configuração varia de programa para programa. Cada programa estabelece o número de gavetas e as gavetas possuem nome, endereço e capacidade de armazenamento diferentes. As gavetas podem armazenar números inteiros, números reais, e caracteres, os quais requerem número de bytes diferentes. O conteúdo das gavetas pode ser lido ou modificado utilizando seu nome ou endereço.

Suponha, por exemplo, que gostaríamos que o computador calculasse a soma de dois números inteiros. Assim como o ser humano, os números são armazenados na memória, são somados, e depois o resultado é armazenado na memória. Para armazenar os números na memória, precisamos estabelecer nome, endereço e capacidade de armazenamento de cada gaveta compatível com um número inteiro. Depois atribuímos os valores para cada gaveta. Como os números não serão mais necessários após a soma, nós podemos usar uma das gavetas para armazenar o resultado. Pedimos

então que o computador execute a soma e armazene o resultado em uma das gavetas. Esses comandos podem ser traduzidos na forma de um algoritmo.

1. Considere as gavetas a e b.
2. Atribua 20 para a e 30 para b.
3. Some a com b e coloque o resultado em a.

Mais especificamente, as gavetas são chamadas **variáveis** e os algoritmos são escritos de uma forma mais elegante.

1. Sejam a e b variáveis inteiras.
2. Faça $a \leftarrow 20$ e $b \leftarrow 30$.
3. Faça $a \leftarrow a + b$.

Este algoritmo pode então ser transcrito na liguagem C da seguinte forma.

```
/* função principal. As funções e alguns comandos têm o escopo
definido entre parênteses. */
int main() {
int a,b;
  a = 20; b = 30;
  a = a + b;
  printf("%d\n",a); /* imprime na tela o resultado. */
}
```

O programa pode ser editado e gravado em um arquivo exemplo.c. O arquivo pode ser compilado usando o comando "gcc exemplo.c -o exemplo" e o arquivo executável "exemplo" será gerado.

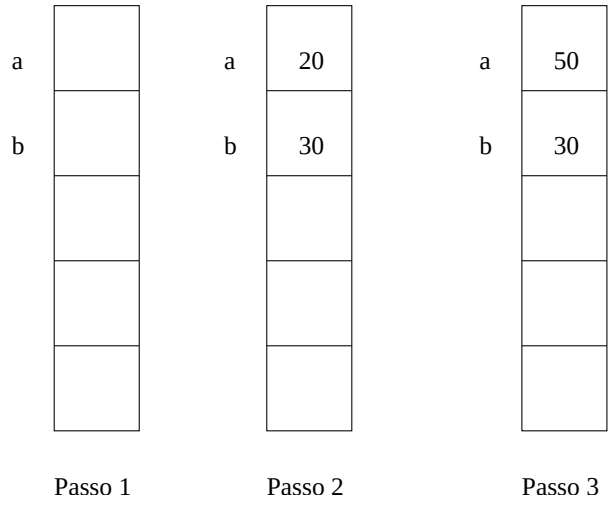

Figura 1.3: Exemplo do jogo das gavetas.

Capítulo 2

Variáveis simples, atribuições e operações matemáticas

2.1 Variáveis simples

Variáveis simples armazenam números e caracteres na memória principal. A forma de armazenamento depende do conteúdo. Variáveis que armazenam números inteiros, por exemplo, consideram 1 bit para o sinal e os demais para o valor do número. Por exemplo, uma variável inteira com b bits pode armazenar valores de $-2^{(b-1)}$ a $2^{(b-1)} - 1$ ou valores sem sinal de 0 a $2^b - 1$. Essas variáveis são declaradas da seguinte forma.

```
int main()
{
int a; /* pode armazenar apenas números inteiros com no máximo 4 bytes,
  incluindo o sinal. Isto é, valores de -2147483648 a 2147483647. */

unsigned int b; /* pode armazenar apenas números inteiros sem sinal com no
     máximo 4 bytes. Isto é, valores de
     0 a 4294967295. */

short c; /* pode armazenar apenas números inteiros com no máximo 2 bytes,
    incluindo o sinal. Isto é, valores de -32768 a 32767. */

unsigned short d; /* pode armazenar apenas números inteiros sem sinal
          com no máximo 2 bytes. Isto é, valores de 0 a
          65535. */

float e; /* pode armazenar números reais com no máximo 4 bytes,
            incluindo o sinal. A forma de armazenamento permite
            representar números nos intervalos de -2 x 10^(-38) a -2 x
            10^(38) e de 2 x 10^(-38) a 2 x 10^(38). */

double f; /* pode armazenar números reais com no máximo 8 bytes,
```

```
            incluindo o sinal. A forma de armazenamento permite
            representar números nos intervalos de -2 x 10^(-308) a -2 x
            10^(308) e de 2 x 10^(-308) a 2 x 10^(308). */

char   g; /* pode armazenar caracteres alfanuméricos com no máximo 1
  byte, incluindo o sinal no caso de número inteiro. Isto é,
  valores como 'a', 'X', '%' e números de -128 a 127.*/

unsigned char   h; /* pode armazenar caracteres alfanuméricos com no máximo 1
  byte, incluindo o sinal no caso de número inteiro. Isto é,
  valores como 'a', 'X', '%' e números de 0 a 255.*/

}
```

Alguns sistemas operacionais consideram apenas 2 bytes para o tipo int. Neste caso, o tipo **long** estende para 4 bytes. O comando **sizeof(tipo)** permite saber quantos bytes são ocupados um dado **tipo** de variável (e.g. $sizeof(int) = 4$ bytes).

Os nomes das variáveis podem ter tamanhos maiores, contendo caracteres alfanuméricos, mas não podem conter caracteres e nem constituírem palavras reservadas pela linguagem para outros fins.

2.2 Atribuições

O operador "=" é usado para indicar uma atribuição de valor à variável. Todo comando de declaração de variável ou de atribuição deve ser finalizado com ";". Exemplos:

```
int main
{
int a;
unsigned int b;
short c;
unsigned short d;
float e;
double f;
char   g;
unsigned char h;

a = 10; /* correto */
b = -6; /* errado */
c = 100000; /* errado */
d = 33000; /* certo */
e = -80000.657; /* certo */
f = 30 /* errado */
g = 'a'; /* certo */
g = a; /* errado, a menos que ``a'' fosse do tipo char */
h = 200; /* certo */
h = 'B'; /* certo */
```

```
}
```

Declarações e atribuições também podem aparecer da seguinte forma:

```
int main()
{
int a=10,b=-30; /* na mesma linha, separadas por ‘‘,’’. */
float c;
char d='4'; /* estou me referindo ao caracter 4 e não ao número. */

c = a; /* converte para float e copia valor 10.0 para ‘‘c’’. */
c = a + 1.8; /* atribui valor 11.8 para ‘‘c’’. */
b = c; /* converte para int truncando a parte decimal e copia 11
  para ‘‘c’’. */
b = a + b; /* soma 10 e 11, e copia 21 para. b */
a = 10 + b; c = b*40.5; /* devemos evitar de escrever vários comenados
   em uma mesma linha. */
}
```

Note que as atribuições só podem ser feitas após a declaração da variável. Podemos também usar um **tipo** de variável entre parênteses, para fazer com que o resultado da expressão seja convertido para o tipo especificado antes da atribuição. Exemplos:

```
int main()
{
int a;

a = (int)(2.5*7.2); /* calcula o produto e depois converte para inteiro. */
}
```

2.3 Operações aritméticas e expressões

Operações aritméticas podem ser combinadas formando expressões. Existe uma regra de prioridade da multiplicação e divisão (mesma prioridade) sobre soma e subtração (mesma prioridade), e a ordem de execução para operadores de mesma prioridade é da esquerda para direita. Exemplos:

```
int main()
{
  int a=20,b=10;
  float c=1.5,d;

  d = c*b/a; /* atribui 0.75 para ‘‘d’’ */
  d = c*(b/a); /* atribui 0.0 para ‘‘d’’, pois a divisão entre
    inteiros resulta em um inteiro. A divisão só resulta
    em número real se ao menos um dos operandos for
    real, porém isto não adianta nada se o resultado for
    atribuído a uma variável inteira. Os parênteses
```

```
    forçam a execução da operação de divisão antes da
    multiplicação. */
  d = b-a*c; /* atribui -20.0 para ‘‘d’’ */
  d = (b-c)*a; /* atribui 170.0 para ‘‘d’’ */
}
```

Note que a melhor forma de garantir o resultado esperado é usar parênteses. A prioridade de execução neste caso é da expressão mais interna para a mais externa.

```
int main()
{
  int a=20,b=10;
  float c=1.5,d;

  d = (((a+5)*10)/2)+b; /* atribui 135 para ‘‘d’’ */
}
```

2.4 Constantes

Assim como variáveis, constantes são usadas para armazenar números e caracteres. Porém, não podemos modificar o conteúdo de uma constante durante a execução do programa. Exemplos:

```
#define PI          3.1415926536 /* atribui 3.1415926536 para PI */
#define MSG        "O Conteúdo de a é " /* atribui o texto para MSG */
#define Epsilon     1E-05    /* atribui 0.00001 para Epsilon */

int main()
{
  float dois_PI;

  dois_PI = 2*PI; /* atribui 6.2831853072 para "a" */
  printf("%s %5.2f\n",MSG,dois_PI); /* imprime "O conteúdo de a é 6.28"
       na tela */
}
```

2.5 Definindo novos tipos

Podemos usar o comando **typedef** para definir novos tipos de variáveis ou abreviar tipos existentes.

```
typedef enum {false,true} bool; /* o tipo bool só armazena 0/false e 1/true */
typedef unsigned char uchar;    /* o tipo uchar é o mesmo que unsigned char */
int main()
{
  bool v1,v2; /* tipo boleano ou variável lógica */
  uchar a=10;

  v1 = true;  /* o mesmo que atribuir 1 para v1 */
```

```
  v2 = false; /* o mesmo que atribuir 0 para v2 */
  printf("%d %d %d\n",v1,v2,a); /* imprime na tela 1 0 10 */
}
```

2.6 Operações lógicas e expressões

Os caracteres "&&", "||", "!" indicam as seguintes operações lógicas: **and**, **or** e **not**, respectivamente. Essas operações quando aplicadas a variáveis lógicas reproduzem os resultados apresentados na Figura 2.1, dependendo do conteúdo das variáveis.

v1	v2	v1 and v2
true	true	true
true	false	false
false	true	false
false	false	false

v1	v2	v1 or v2
true	true	true
true	false	true
false	true	true
false	false	false

v1	not v1
true	false
false	true

Figura 2.1: Operações lógicas

```
typedef enum {false,true} bool;
int main()
{
  bool v1,v2,v3;

  v1 = true;
  v2 = false;
  v3 = v1&&v2; /* atribui false para ‘‘v3’’ */
  v3 = v1||v2; /* atribui true para ‘‘v3’’ */
  v3 = !v1;    /* atribui false para ‘‘v3’’ */
  v3 = ((v1&&v2)||(!v2)); /* atribui true para ‘‘v3’’ */
}
```

2.7 Funções matemáticas e resto da divisão inteira

Note que um programa consiste de uma seqüência de comandos apresentada na **função principal** (*main*). Quando várias tarefas são necessárias, as seqüências de comandos dessas tarefas podem ser agrupadas em outras **funções**. Esta forma de organização dos programas evita confusão na depuração

do programa e provê uma melhor apresentação do código fonte. Estas funções podem ser chamadas a partir da função principal e devem retornar o resultado da tarefa correspondente, para que as demais tarefas possam ser executadas. A própria função main deve retornar o valor 0, quando termina com sucesso. Várias funções matemáticas, por exemplo, são disponibilizadas para o programador na linguagem C.

```
#include <math.h> /* Para usar as funções matemáticas precisamos
     incluir suas definições, que estão em math.h. */

#define PI 3.1415926536

int main()
{
  double a,b;
  int c,d,e;

  a = 1.0;
  b = exp(a); /* atribui 2.718282 para b */
  a = 4.0;
  a = pow(a,3.0); /* atribui 64.0 para a */
  b = log10(100); /* atribui 2.000000 para b */
  a = sin(PI/4.0); /* atribui 0.707107 para a */
  c = 5;
  d = 3;
  e = c/d; /* atribui 1 para c - divisão inteira */
  e = c%d; /* atribui 2 para e - resto da divisão inteira */

  return(0);
}
```

Além de incluir as definições de math.h, as funções matemáticas precisam ser compiladas e incorporadas ao programa executável. Isto é feito com o comando "gcc exemplo.c -o exemplo -lm". Outros exemplos de funções matemáticas são: raíz quadrada **sqrt(x)**, cosseno **cos(x)**, arco tangente **atan(x)**, logarítmo Neperiano **ln(x)**, e arredondamento **round(x)**. Essas funções podem ser encontradas em qualquer manual da linguagem C ou através do comando **man** para aqueles que usam linux.

2.8 Endereço das variáveis

Toda variável x tem um endereço na memória que pode ser acessado com $\&x$. Este endereço é necessário em algumas funções de entrada de dados, que veremos a sequir.

2.9 Exercícios

Sabendo que as fórmulas de conversão de temperatura de Celsius para Fahrenheit e vice-versa são

$$C = \frac{(F-32)5}{9}, \tag{2.1}$$

$$F = \frac{9C}{5} + 32, \tag{2.2}$$

escreva dois programas em C, um para converter de Celsius para Fahrenheit e outro para fazer o caminho inverso.

Capítulo 3

Entrada e saída de dados

Comandos de entrada e saída são usados para fornecer números e caracteres que serão armazenados em variáveis na memória primária. Vamos aprender, inicialmente, apenas os comandos **scanf** e **printf** de leitura e escrita, respectivamente.

3.1 Entrada padrão

O teclado é o dispositivo padrão para entrada de dados. O comando scanf ler números e caracteres digitados no teclado da seguinte forma:

```
scanf("%d",&a);
```

onde "a" é uma variável inteira e "&a" é o endereço em memória da variável "a".

Cada tipo de variável requer um símbolo específico de conversão. Por exemplo, "%d" indica conversão da entrada para um valor inteiro. O símbolo "%d" pode ser usado com variáveis int, short, unsigned short, unsigned char; o símbolo "%u" é usado para unsigned int; o símbolo "%c" para char; o símbolo "%s" para char e cadeias de caracteres; o símbolo "%f" para float; e o símbolo "%lf" para double. Exemplo:

```
#include<stdio.h> /* Deve ser incluída para scanf e printf */

int main()
{
  int  a;
  float b;
  char c;
  double d;
  unsigned int e;
  unsigned char f;
  short g;
  unsigned short h;

  printf("Digite um inteiro: ");
  scanf("%d",&a);
```

```
  printf("Digite um float: ");
  scanf("%f",&b);
  printf("Digite um caracter: ");
  scanf(" %c",&c); /* %c só funcionou com espaço em branco na
      frente. Outra opção é usar %s */
  printf("Digite um double: ");
  scanf("%lf",&d);
  printf("Digite um inteiro sem sinal: ");
  scanf("%u",&e);
  printf("Digite um inteiro sem sinal de 0 a 255: ");
  scanf("%d",&f);
  printf("Digite um short: ");
  scanf("%d",&g);
  printf("Digite um short sem sinal: ");
  scanf("%d",&h);
  return(0);
}
```

3.2 Saída padrão

A tela é o dispositivo padrão para saída de dados. O comando printf apresenta números e caracteres na tela da seguinte forma:

```
printf("%d",a); /*  Mostra o conteúdo da variável ‘‘a’’. */
```

A simbologia de conversão é a mesma usada no scanf. Esta saída também pode ser formatada indicando o número de dígitos do número.

```
#include<stdio.h> /* Deve ser incluída para scanf e printf */

int main()
{
  int a=9,b=800;
  float c=20.9176,d=-1.4219;

  /* \n pula linha, 3 indica o número de dígitos (incluindo o sinal,
     se houver), 5 indica o número de dígitos total (incluindo o
     sinal, se houver) e 2 o número de dígitos após o ponto. */

  printf("\n a=%3d,\n b=%3d,\n c=%5.2f, \n d=%5.2f.\n",a,b,c,d);
  return(0);
}
```

A saída do programa será:

```
a=  9, /* alinha pela direita com espaços em branco à esquerda do número. */
b=800,
c= 20.92, /* arredonda para cima. */
d= -1.42. /* arredonda para baixo. */
```

3.3 Exercícios

1. Escreva um programa para converter temperaturas de Celsius para Farenheit usando a entrada padrão e a saída padrão para obter e mostrar as temperaturas, respectivamente. Repita o exercício para converter de Farenheit para Celsius.

2. Escreva um programa para trocar os conteúdos de duas variáveis inteiras, x e y, usando uma terceira variável z. Leia os valores de x e y da entrada padrão e apresente os resultados na saída padrão. Repita o exercício sem usar a terceira variável z.

Capítulo 4

Comando condicional

Em muitas tarefas de programação, desejamos que o computador execute instruções diferentes, dependendo de alguma condição lógica. Por exemplo, no cálculo das raízes de uma equação de segundo grau, teremos instruções diferentes para raízes imaginárias e para raízes reais.

4.1 Estrutura condicional simples

Um comando condicional simples é aquele que permite a escolha de um grupo de instruções (**bloco de comandos**) quando uma determinada condição lógica é satisfeita.

```
if (expressão) {
  bloco de comandos
}
```

```
#include<stdio.h>

int main()
{
  int a,b;

  printf("Digite a e b: ");

  scanf("%d %d",&a,&b); /* podemos ler mais de uma variável por linha,
   deixando um espaço em branco entre os números. */

  if (a > b) { /* O bloco de comandos deve ser delimitado por
  chaves. Neste caso, como temos um único comando, as
  chaves são opcionais. A instrução só será executada,
  caso o valor de ‘‘a’’ seja maior que o valor de
  ‘‘b’’ (ou b < a). */
   printf("%d é maior que %d\n",a,b);
  }

  if (a <= b) /* ou b >= a */
```

```
    printf("%d é menor ou igual a %d\n",a,b);

  if (a == b) /* NUNCA FAÇA a=b */
    printf("%d é igual a %d\n",a,b);

  if (a != b)
    printf("%d é diferente de %d\n",a,b);

  return(0);
}
```

A condição lógica também pode ser resultado de uma expressão lógica mais complicada.

```
#include<stdio.h>

int main()
{
  int a,b,c;

  printf("Digite a, b, e c: ");
  scanf("%d %d %d",&a,&b,&c);

  if ((a > b)&&(b > c))
   printf("%d é maior que %d e %d\n",a,b,c);

  if (((a == b)&&(b < c))||
      ((a == c)&&(c < b))){
    printf("%d é igual a %d e menor que %d\n",a,b,c);
    printf("ou %d é igual a %d e menor que %d\n",a,c,b);
  }
  return(0);
}
```

4.2 Estrutura condicional composta

Um comando condicional composto é aquele que permite a escolha de um bloco de comandos, quando uma determinada condição é satisfeita, e de um outro bloco de comandos quando a condição não é satisfeita.

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

#include<stdio.h>
```

```
int main()
{
  int a,b;

  printf("Digite a e b: ");
  scanf("%d %d",&a,&b);

  if (a > b) {
   printf("%d é maior que %d\n",a,b);
  } else { /* Essas chaves são opcionais, pois o bloco
      tem um único comando.*/
   printf("%d é menor ou igual a %d\n",a,b);
  }
  return(0);
}
```

Note que, um bloco de comandos pode ter outros comandos condicionais.

```
#include<stdio.h>

int main()
{
  int a,b,c;

  printf("Digite a, b, e c: ");
  scanf("%d %d %d",&a,&b,&c);

  if (a > b) { /* Neste caso, as chaves são obrigatórias. */
   printf("%d é maior que %d\n",a,b);
   if (b > c)
     printf("%d é maior que %d\n",b,c);
   else /* b <= c */
     printf("%d é menor ou igual a %d\n",b,c);
  } else { /* a <= b */
   printf("%d é menor ou igual a %d\n",a,b);
   if (b < c)
     printf("%d é menor que %d\n",b,c);
   else /* b >= c */
     printf("%d é maior ou igual a %d\n",b,c);
  }

  return(0);
}
```

4.3 Exercícios

1. Escreva um único programa para converter entre Celsius e Farenheit, e vice-versa.
2. Escreva um programa para calcular o maior número entre três números lidos.

Capítulo 5

Comandos de repetição

Em muitas tarefas de programação, desejamos que um bloco de comandos seja executado repetidamente até que determinada condição seja satisfeita.

5.1 Comando do while

O comando **do while** é uma instrução de repetição, onde a condição de interrupção é testada após executar o comando.

```
do {
bloco de comandos
} while (expressão lógica);
```

O bloco de comandos é repetido até que a expressão seja falsa. Suponha, por exemplo, um programa para dizer se um número inteiro lido da entrada padrão é par ou ímpar. Desejamos que o programa execute até digitarmos um número negativo.

```
#include <stdio.h>

int main()
{
  int n;

  do {

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

    if ((n%2)==0)
      printf("O número é par\n");
    else
      printf("O número é ímpar\n");

  } while (n >= 0);
```

```
  return(0);
}
```

Um exemplo prático interessante é o algoritmo de Euclides para calcular o Máximo Divisor Comum (MDC) de dois números inteiros positivos.

1. Leia m e n.
2. Faça $x \leftarrow m$ e $y \leftarrow n$.
3. Atribua a r o resto da divisão de x por y.
4. Faça $x \leftarrow y$ e $y \leftarrow r$.
5. Volte para a linha 3 enquanto $r \neq 0$.
6. Diga que x é o MDC de m e n.

Codifique este algoritmo em C.

5.2 Comando while

O comando **while** é uma instrução de repetição, onde a expressão lógica é testada antes de executar o comando. Sua estrutura básica envolve quatro etapas: inicialização de uma variável de controle, teste de interrupção envolvendo a variável de controle, execução do bloco de comandos, e atualização da variável de controle.

```
inicialização
while(expressão lógica)
{
  bloco de comandos
  atualização
}
```

Suponha, por exemplo, um programa que soma n valores reais lidos da entrada padrão e apresenta o resultado na saída padrão.

```
#include <stdio.h>

int main()
{
  int i,n; /* variável de controle i */
  float soma,num;

  printf("Entre com a quantidade de números a serem somados: "); scanf("%d",&n);
  /* inicialização */
  i = 1; soma = 0.0;
```

```
  while (i <= n) {   /* expressão */
    /* bloco de comandos */
    printf("Digite o %do. número:",i);
    scanf("%f",&num);
    soma = soma + num;
    i    = i + 1; /* atualização */
  }
  printf("O resultado da soma é %f\n",soma);

  return(0);
}
```

Modifique o programa acima para ele calcular o produto dos n números.

Observe que nos exemplos envolvendo o comando do while, a variável de controle é inicializada e atualizada no bloco de comandos. Neste caso, a construção com do while fica mais elegante do que com o comando while. Por outro lado, em situações onde a variável de controle não participa do bloco principal de comandos, a construção fica mais elegante com o comando while.

Podemos combinar vários comandos if, do while e while aninhados.

```
#include <stdio.h>

int main()
{
  int i,n;
  float soma,num;
  char opt;

  do {
    printf("Digite a opção desejada:\n");
    printf("s para somar números, \n");
    printf("ou qualquer outra letra para sair do programa.\n");
    scanf(" %c",&opt);
    if (opt == 's'){
      printf("Entre com a quantidade de números a serem somados\n");
      scanf("%d",&n);
      i = 1; soma = 0.0;
      while (i <= n) {
printf("Digite o %do. número:",i); scanf("%f",&num); soma = soma + num;
i    = i + 1;
      }
      printf("O resultado da soma é %f\n",soma);
    }
  } while (opt == 's');

  return(0);
}
```

Podemos também criar um laço infinito, substituindo **do while** por **while(1)**, e sair dele com o comando **break** no **else** do **if**. Outro comando de desvio interessante é o comando **continue**. Ele permite desconsiderar o restante do bloco de comandos, voltando para o teste da expressão lógica. Por exemplo, podemos desprezar o processamento de números negativos, acrescentando:

```
if (num < 0) continue;
```

após a leitura do número.

5.3 Comando for

O comando **for** é uma simplificação do comando **while**, onde a inicialização da variável de controle, a expressão lógica envolvendo a variável de controle e a atualização da variável são especificadas no próprio comando. Sua implementação é feita com o comando while, portanto seu comportamento é o mesmo: após a inicialização, a expressão lógica é testada. Se for verdadeira, o bloco de comandos é executado. Após execução, a variável de controle é atualizada, a expressão lógica é verificada, e o processo se repete até que a expressão seja falsa.

```
for (inicialização; expressão; atualização)
{
bloco de comandos
}
```

Por exemplo, um programa para somar n números fica.

```
#include<stdio.h>

int main()
{
  int n,i;
  float soma,num;

  printf("Entre com a quantidade de números a serem somados: ");
  scanf("%d",&n);

  for (i=1, soma = 0.0; i <= n; i=i+1, soma = soma + num) {
    printf("Digite o %do. número:",i);
    scanf("%f",&num);
  }
  printf("O resultado da soma é %f\n",soma);

  return(0);
}
```

Uma observação interessante é que a sintaxe para incrementar/decrementar e multiplicar/dividir valores permite as seguintes variações.

```
soma += num; /* É o mesmo que soma = soma + num; */
prod *= num; /* É o mesmo que prod = prod * num; */
y /= 2;      /* É o mesmo que y = y / 2; */
i++;         /* É o mesmo que i = i + 1; */
i--;         /* É o mesmo que i = i - 1; */
```

Outro exemplo é um programa para calcular o maior entre n números lidos da entrada padrão.

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

int main()
{
  int i,n,num,maior;

  printf("Entre com a quantidade de números: ");
  scanf("%d",&n);

  maior = INT_MIN;
  for(i=1; i <= n; i++) {
    printf("Entre com um inteiro: ");
    scanf("%d",&num);
    if (num > maior)
      maior = num;
  }
  printf("O maior inteiro lido foi %d\n",maior);
  return(0);
}
```

Sabendo que o fatorial de um número inteiro n é $n \times (n-1) \times (n-2) \ldots 1$, faça um programa para calcular o fatorial de um número lido da entrada padrão usando o comando **for** e apenas duas variáveis.

O triângulo de Floyd é formado por n linhas de números consecutivos, onde cada linha contém um número a mais que a linha anterior. Para imprimir o triângulo de Floyd precisamos aninhar um for dentro do outro.

```
#include <stdio.h>

int main()
{
  int l,c,nl,i;

  printf("Entre com o número de linhas: ");
  scanf("%d",&nl);

  i = 1;
  for(l=1; l <= nl; l++) {
```

```
    for(c=1; c <= l; c++) {
      printf("%2d ",i);
      i++;
    }
    printf("\n");
  }

  return(0);
}
```

Outro exemplo que requer dois comandos for aninhados é a impressão de uma tabuada. Faça um programa para imprimir uma tabuada com n linhas e n colunas.

Observe que existe uma relação entre a integral de uma função contínua, sua aproximação pela somatória de uma função discreta, e a implementação da somatória usando o comando for. Por exemplo,

$$\int_{x_{\min}}^{x_{\max}} (ax+b)d_x \quad \approx \quad \sum_{x=x_{\min}+d_x/2}^{x=x_{\max}-d_x/2} (ax+b)d_x, \tag{5.1}$$

onde $x_{\min} < x_{\max}$, é uma aproximação para a integral da curva, a qual tem maior exatidão para valores menores de $d_x > 0$. Podemos implementar esta integral como:

```
#include<stdio.h>

int main()
{
  float a,b,xmin,xmax,x,dx,integral;

  printf("Entre com os coeficientes a e b da reta y=ax+b: \n");
  scanf("%f %f",&a,&b);

  printf("Entre com o intervalo [xmin,xmax] para cálculo de área: \n");
  scanf("%f %f",&xmin,&xmax);

  printf("Entre com o incremento dx: \n");
  scanf("%f",&dx);

  integral = 0.0;
  for (x=xmin+dx/2.0; x <= xmax-dx/2.0; x=x+dx)
    integral += (a*x+b)*dx;
  printf("integral %f\n",integral);

  return(0);
}
```

5.4 Exercícios

Altere o programa que calcula o MDC de dois números para usar o comando while, e faça com que ele se repita até digitarmos algo diferente de 'm'.

Capítulo 6

Comando Switch

Frequentemente desejamos evitar aninhar vários comandos **if** para tratar múltiplas condições. O comando **switch** faz este papel verificando o conteúdo de variáveis dos tipos int, char, unsigned int, unsigned short, short, e unsigned char.

```
switch (variável) {
case conteúdo1:
  bloco de comandos
  break;
case conteúdo2:
  bloco de comandos
  break;
.
.
.
case conteúdon:
  bloco de comandos
  break;
default:
  bloco de comandos
}
```

Suponha, por exemplo, um programa que faz o papel de uma calculadora.

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

#define PI 3.1415926536

int main()
{
  double a,b,c;
  char opt;
```

```
do {
  printf("Digite a opção desejada\n");
  printf("*: multiplicação\n");
  printf("/: divisão\n");
  printf("+: adição\n");
  printf("-: subtração\n");
  printf("r: raiz quadrada\n");
  printf("p: potência\n");
  printf("t: tangente\n");
  printf("a: arco tangente\n");
  printf("x: sair do programa\n");

  scanf(" %c",&opt);

  switch(opt) {
  case '*':
    printf("Digite os dois operandos: ");
    scanf("%lf %lf",&a,&b);
    c = a*b;
    printf("\n\n %8.4lf*%8.4lf=%8.4lf\n\n",a,b,c);
    break;
  case '/':
    printf("Digite os dois operandos: ");
    scanf("%lf %lf",&a,&b);
    if (b!=0.0){
      c = a/b;
      printf("\n\n %8.4lf/%8.4lf=%8.4lf\n\n",a,b,c);
    }else{
      printf("\n\n OperaÃ§Ã£o invÃ<lida\n\n");
    }
    break;
  case '+':
    printf("Digite os dois operandos: ");
    scanf("%lf %lf",&a,&b);
    c = a+b;
    printf("\n\n %8.4lf+%8.4lf=%8.4lf\n\n",a,b,c);
    break;
  case '-':
    printf("Digite os dois operandos: ");
    scanf("%lf %lf",&a,&b);
    c = a-b;
    printf("\n\n %8.4lf-%8.4lf=%8.4lf\n\n ",a,b,c);
    break;
  case 'r':
    printf("Digite o número: ");
    scanf("%lf",&a);
```

```
    if (a >= 0){
      c=sqrt(a);
      printf("\n\n sqrt(%8.4lf)=%8.4lf\n\n",a,c);
    }else{
      c=sqrt(-a);
      printf("\n\n sqrt(%8.4lf)=%8.4lf i\n\n",a,c);
    }
    break;
  case 'p':
    printf("Digite os dois operandos: ");
    scanf("%lf %lf",&a,&b);
    c = pow(a,b);
    printf("\n\n pow(%8.4lf,%8.4lf)=%8.4lf\n\n ",a,b,c);
    break;
  case 't':
    printf("Digite o ângulo em graus: ");
    scanf("%lf",&a);
    if (((int)a)%90==0.0)
    {
      b = ((int)a)%360;
      if (b < 0.0)
        b = 360.0 + b;
      if (b==270)
        printf("\n\n tan(%f)=-infinito\n\n",a);
      else
        printf("\n\n tan(%f)=+infinito\n\n",a);
    }else{
      b = PI*a/180.0;
      c = tan(b);
      printf("\n\n tan(%8.4lf)=%8.4lf\n\n ",a,c);
    }
    break;
  case 'a':
    printf("Digite o arco: ");
    scanf("%lf",&a);
    c = atan(a);
    c = c*180.0/PI;
    printf("\n\n atan(%8.4lf)=%8.4lf graus\n\n",a,c);
    break;
  case 'x':
    break;
  default:
    printf("\n\n Opção inválida\n\n");
  }
} while (opt != 'x');
```

```
  return(0);
}
```

Continue o programa acrescentando outras funções.

Outro exemplo é a geração de menus de programas que envolvem vários comandos switch aninhados.

```
#include <stdio.h>

int main()
{
  char opt1,opt2;

  do
  {
    printf("Digite a opção desejada:\n");
    printf("0 - Conta corrente\n");
    printf("1 - Aplicações\n");
    printf("2 - Cancelar a operação\n");
    scanf(" %c",&opt1);

    switch(opt1) {

    case '0':

      do {
        printf("Digite a opção desejada:\n");
        printf("0 - Saldo da conta corrente\n");
        printf("1 - Extrato da conta corrente\n");
        printf("2 - Saque da conta corrente\n");
        printf("3 - Voltar para o menu principal\n");
        printf("4 - Cancelar operação\n");
        scanf(" %c",&opt2);

      switch(opt2) {
      case '0':
        printf("\n\n Imprime saldo\n\n");
        break;
      case '1':
        printf("\n\n Imprime extrato\n\n");
        break;
      case '2':
        printf("\n\n Digite o valor do saque\n\n");
        break;
      case '3':
        break;
      case '4':
```

```
      opt2='3';
      opt1='2';
    break;
    default:
      printf("\n\n Opção inválida\n\n");
    }
  } while(opt2 != '3');
  break;

case '1':

  do {
    printf("Digite a opção desejada:\n");
    printf("0 - Saldo da aplicação\n");
    printf("1 - Extrato da aplicação\n");
    printf("2 - Saque da aplicação\n");
    printf("3 - Voltar para o menu principal\n");
    printf("4 - Cancelar operação\n");
    scanf(" %c",&opt2);

    switch(opt2) {
    case '0':
      printf("\n\n Imprime saldo\n\n");
      break;
    case '1':
      printf("\n\n Imprime extrato\n\n");
      break;
    case '2':
      printf("\n\n Digite o valor do saque\n\n");
      break;
    case '3':
      break;
    case '4':
      opt2='3';
      opt1='2';
    break;
    default:
      printf("\n\n Opção inválida\n\n");
    }
  } while(opt2 != '3');
  break;

 case '2':
   break;

 default:
```

```
      break;
    }
   } while (opt1!='2');

return(0);
}
```

Capítulo 7

Vetores

Vimos que uma variável simples está associada a uma posição de memória e qualquer referência a ela significa um acesso ao conteúdo de um pedaço de memória, cujo tamanho depende de seu tipo. Nesta aula iremos ver um dos tipos mais simples de estrutura de dados, denominada **vetor**, que nos possibilitará associar um identificador a um conjunto de variáveis simples de um mesmo tipo. Naturalmente, precisaremos de uma sintaxe apropriada para acessar cada variável deste conjunto de forma precisa.

```
int main()
{
  tipo identificador[número de variáveis];
}
```

Antes de iniciarmos, considere que desejamos ler 10 notas de alunos e imprimir as notas acima da média. Note que para executar a tarefa, precisamos calcular a média primeiro e depois comparar cada nota com a média. Portanto, precisamos armazenar cada nota em uma variável. Agora imagine que a turma tem 100 alunos. Como criar 100 variáveis sem tornar o acesso a elas algo complicado de se programar?

7.1 Vetores

Um **vetor** é um conjunto de posições consecutivas de memória, identificadas por um mesmo nome, individualizadas por índices e cujo conteúdo é do mesmo tipo. Assim, o conjunto de 10 notas pode ser associado a apenas um identificador, digamos nota, que passará a identificar não apenas uma única posição de memória, mas 10.

```
int main()
{
  float nota[10]; /* vetor de 10 variáveis do tipo float */
}
```

A referência ao conteúdo da n-ésima variável é indicada pela notação nota[$n-1$], onde n é uma expressão inteira ou uma variável inteira.

A nota de valor 7.0 na Figura 1, que está na quarta posição da seqüência de notas é obtida como nota[3]. Assim, um programa para resolver o problema acima fica:

```
#include <stdio.h>

int main()
{
  float nota[10],media=0.0;
  int i;

  printf("Entre com 10 notas\n");
  for (i=0; i < 10; i++) {
    scanf("%f",&nota[i]);
    media += nota[i];
  }
  media /= 10.0;
  printf("media %5.2f\n",media);

  for (i=0; i < 10; i++) {
    if (nota[i] > media)
      printf("nota[%d]=%5.2f\n",i,nota[i]);
  }

  return 0;
}
```

2.5	4.5	9.0	7.0	5.5	8.3	8.7	9.3	10.0	9.0
0	1	2	3	4	5	6	7	8	9

Figura 7.1: Vetor de 10 notas representado pelo identificador nota.

7.2 Busca em Vetores

Um problema comum quando se manipula vetores é a necessidade de encontrar um elemento com um dado valor. Uma forma trivial de fazer este acesso é percorrer do índice inicial ao índice final todos os elementos do vetor até achar o elemento desejado. Esta forma de busca é chamada linear, pois no pior caso o número de comparações necessárias é igual ao número de elementos no vetor.

7.2.1 Busca Linear

Suponha, por exemplo, que desejamos saber se existe uma nota x no vetor lido.

```
#include <stdio.h>

int main()
{
  float nota[11],x; /* vetor criado com uma posição a mais */
```

```
  int i;

  printf("Entre com 10 notas\n");
  for (i=0; i < 10; i++) {
    scanf("%f",&nota[i]);
  }

  while(1) {
    printf("Digite a nota procurada ou -1 para sair do programa\n");
    scanf("%f",&x);

    if (x==-1.0)
      break;

    /* busca linear */

    nota[10] = x; /* elemento sentinela */
    i = 0;
    while (nota[i] != x) /* busca com sentinela */
      i++;

    if (i < 10)
      printf("nota %5.2f encontrada na posição %d\n",nota[i],i);
    else
      printf("nota %5.2f não encontrada\n",x);
  }

  return 0;
}
```

Imagine agora que nosso vetor tem tamanho 1024. O que podemos fazer para reduzir o número de comparações? Quanto maior for a quantidade de informação sobre os dados, mais vantagens podemos tirar para agilizar os algoritmos.

7.2.2 Busca Binária

A busca binária ,por exemplo, reduz o número de comparações de n para $\log_2(n)$ no pior caso, onde n é o tamanho do vetor. Ou seja, um vetor de tamanho $1024 = 2^{10}$ requer no pior caso 10 comparações. No entanto, a busca binária requer que o vetor esteja ordenado. Esta ordenação também tem um custo a ser considerado, mas se vamos fazer várias buscas, este custo pode valer a pena. A idéia básica é que a cada iteração do algoritmo, podemos eliminar a metade dos elementos no processo de busca. Vamos supor, por exemplo, que o vetor de notas está em ordem crescente.

```
#include <stdio.h>

typedef enum {false,true} bool;
```

```
int main()
{
  float nota[10],x;
  int i,pos,inicio,fim;
  bool achou;

  printf("Entre com 10 notas em ordem crescente\n");
  for (i=0; i < 10; i++) {
    scanf("%f",&nota[i]);
  }

  while(1) {
    printf("Digite a nota procurada ou -1 para sair do programa\n");
    scanf("%f",&x);

    if (x==-1.0)
      break;

    /* busca binária */

    inicio = 0;
    fim    = 9;
    achou  = false;

    while ((inicio <= fim)&&(!achou)){
      pos = (inicio+fim)/2;
      if (x < nota[pos])
        fim = pos-1;
      else
        if (x > nota[pos])
          inicio = pos + 1;
        else
          achou = true;
    }

    if (achou)
      printf("nota %5.2f encontrada na posição %d\n",nota[pos],pos);
    else
      printf("nota %5.2f não encontrada\n",x);
  }

  return 0;
}
```

7.3 Ordenação

Em muitas situações, tal como na busca binária, nós desejamos ordenar vetores. Nesta aula vamos aprender três algoritmos básicos de ordenação: ordenação por seleção, ordenação por inserção e ordenação por permutação. Suponha, por exemplo, que desejamos colocar em ordem crescente o vetor de 10 notas da aula anterior.

7.3.1 Ordenação por seleção

O algoritmo mais intuitivo é o de ordenação por seleção (*selection sort*).A idéia básica é percorrer o vetor várias vezes, selecionando o maior elemento, e trocando-o com o da última posição ainda não usada.

```
#include <stdio.h>

int main()
{
  float nota[10];
  int i,j,jm;

  printf("Entre com 10 notas\n");
  for (i=0; i < 10; i++) {
    scanf("%f",&nota[i]);
  }

  /* Coloque-as em ordem crescente por Seleção */

  for (j=9; j >= 2; j--){
  /* seleciona a maior nota entre j notas e troca-a com a da posição j */
    jm    = j;
    for (i=0; i < j; i++){
      if (nota[i] > nota[jm]){
        jm    = i;
      }
    }

  /* troca */
    if (j != jm){
      nota[j]  = nota[j] + nota[jm];
      nota[jm] = nota[j] - nota[jm];
      nota[j]  = nota[j] - nota[jm];
    }
  }

  /* imprime as notas ordenadas */

  for (i=0; i < 10; i++) {
```

```
    printf("%f ",nota[i]);
  }
  printf("\n");

  return 0;
}
```

Observe que no pior caso teremos $\frac{n(n-1)}{2}$ comparações entre notas e $n-1$ trocas, onde n é o tamanho do vetor. Dizemos então que o algoritmo executa em tempo proporcional ao quadrado do número de elementos e adotamos a notação $O(n^2)$. O processo de ordenação também não usa nenhum vetor auxiliar, portanto dizemos que a ordenação é *in place*. Esta observação se aplica aos outros métodos abaixo.

7.3.2 Ordenação por inserção

A ordenação por inserção (*insertion sort*) se assemelha ao processo que nós usamos para ordenar cartas de um baralho. A cada passo nós ordenamos parte da seqüência e a idéia para o passo seguinte é inserir a próxima nota na posição correta da seqüência já ordenada no passo anterior.

```
#include <stdio.h>

int main()
{
  float nota[10];
  int i,j;

  printf("Entre com 10 notas\n");
  for (i=0; i < 10; i++) {
    scanf("%f",&nota[i]);
  }

  /* Coloque-as em ordem crescente por Inserção */

  for (j=1; j < 10; j++){
    /* insere a nota da posição j na posição correta da
       seqüência já ordenada da posição 0 até j-1 */
    i = j;
    while ((i > 0)&&(nota[i] < nota[i-1])){
    /* troca as notas das posições i e i-1 */
      nota[i]   = nota[i] + nota[i-1];
      nota[i-1] = nota[i] - nota[i-1];
      nota[i]   = nota[i] - nota[i-1];
      i--;
    }
  }
```

```
  /* imprime as notas ordenadas */

  for (i=0; i < 10; i++) {
    printf("%f ",nota[i]);
  }
  printf("\n");

  return 0;
}
```

Observe que no pior caso, o número de comparações e de trocas é $\frac{n(n-1)}{2}$, onde n é o tamanho do vetor. Portanto, o algoritmo é também $O(n^2)$, apesar do número maior de trocas.

7.3.3 Ordenação por permutação

A idéia básica é simular o processo de ebolição de uma bolha de ar dentro d'água (*bubble sort*). A cada passo, a maior nota, que representa a bolha, deve subir até a superfície. Ou seja, trocas são executadas do início para o final do vetor, até que a maior nota fique na última posição ainda não usada.

```
#include <stdio.h>

int main()
{
  float nota[10];
  int i,j;

  printf("Entre com 10 notas\n");
  for (i=0; i < 10; i++) {
    scanf("%f",&nota[i]);
  }

  /* Coloque-as em ordem crescente por Permutação */

  for (j=9; j > 0; j--){
    /* seleciona a maior nota entre cada par de notas consecutivas,
       faz a troca se necessário, até que a maior nota seja inserida
       na posição j+1 */
    for (i=0; i < j; i++) {
      if (nota[i] > nota[i+1]){ /* troca as notas */
        nota[i]   = nota[i+1] + nota[i];
        nota[i+1] = nota[i]   - nota[i+1];
        nota[i]   = nota[i]   - nota[i+1];
      }
    }
  }
```

```
  /* imprime as notas ordenadas */

  for (i=0; i < 10; i++) {
    printf("%f ",nota[i]);
  }
  printf("\n");

  return 0;
}
```

No pior caso, temos também $\frac{n(n-1)}{2}$ comparações e trocas, onde n é o tamanho do vetor. Portanto, o algoritmo é $O(n^2)$ com número de trocas equivalente ao da ordenação por inserção.

7.4 Operações com vetores

Uma das aplicações para vetores é a representação de sinais e funções discretas. Em telecomunicações, por exemplo, é muito comum obter amostras de um sinal elétrico, que representa um trecho de um sinal de voz, e armazená-las em um vetor. O processamento do sinal no computador é essencialmente uma seqüência de operações aplicadas ao vetor. O sinal processado pode então ser transformado de volta em sinal elétrico, e por sua vez transformado em som.

7.4.1 Reflexão

Seja $f_1(x)$ um sinal discreto definido no intervalo de inteiros $[0, n) = 0, 1, \ldots, n-1$. Assumimos que $f_1(x) = 0$ fora deste intervalo. A reflexão $f_2(x) = f_1(-x)$ do sinal em torno da origem é um sinal definido no intervalo $(-n, 0]$, com valores nulos fora deste intervalo. Muito embora as amostras desses sinais estejam definidas em intervalos diferentes do eixo x, ambos são armazenados em vetores com n posições, cujos índices variam de 0 a $n-1$. As diferenças entre eles são os valores das amostras para cada índice i, $i = 0, 1, \ldots, n-1$, e a relação entre i e x: $i = x$ para f_1 e $i = -x$ para f_2. Portanto, para refletir um sinal em torno da origem é só inverter a ordem dos elementos do vetor. Note que isto independe da relação entre i e x (i.e. da localização da origem $x = 0$).

```
#include <stdio.h>

#define N 100

int main()
{
  float f1[N],f2[N];
  int i,n;

  printf("Entre com o número de amostras\n");
  scanf("%d",&n);
  printf("Entre com os valores das amostras\n");
  for (i=0; i < n; i++)
    scanf("%f",&f1[i]);
```

```
  /* calcula a reflexão */

  for (i=0; i < n; i++)
    f2[n-1-i] = f1[i];

  printf("Vetor resultante\n");
  for (i=0; i < n; i++)
    printf("%5.2f ",f2[i]);
  printf("\n");

  return(0);
}
```

7.4.2 Convolução

A **convolução** entre dois sinais discretos $f_1(x)$, $x \in [0, n_1)$, e $f_2(x)$, $x \in [-n_2/2, n_2/2]$, é um terceiro sinal discreto $f_3(x)$, $x \in [-n_2/2, n_1 + n_2/2)$, com $n_1 + n_2 - 1$ amostras.

$$f_3(x) = \sum_{x'=-\infty}^{x'=+\infty} f_1(x') f_2(x - x') \tag{7.1}$$

Observe que para facilitar a visualização do processo, podemos manter a função $f_1(x')$ fixa, mas deslocada $n_2/2$ amostras para a direita, e deslizar a função $f_2(x - x')$ da esquerda para a direita para cada valor de $x = 0, 1, \ldots, n_1 + n_2 - 2$ (ver Figura 7.2).

$$f_3(x - n_2/2) = \sum_{x'=0}^{x'=n_1+n_2-2} f_1(x' - n_2/2) f_2(x - x'). \tag{7.2}$$

Assim, o índice i do vetor que armazena $f_3(x)$ tem relação $i = x - n_2/2$.

A convolução pode ser usada para filtrar o sinal, suavizando transições abruptas (e.g. $f_2(x) = \{1, 2, 1\}$), detectando transições abruptas (e.g. $f_2(x) = \{-1, 2, -1\}$ detecta transições como cruzamentos da função com o eixo x), ou realçando essas transições (e.g. $f_2(x) = \{1, 0, -1\}$).

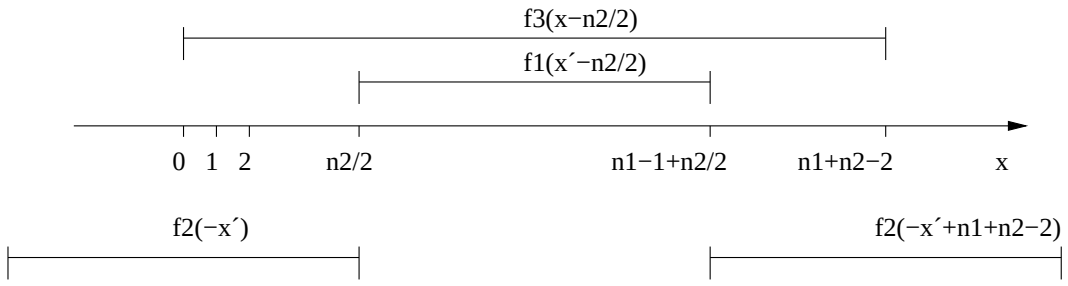

Figura 7.2: Convolução entre sinais discretos.

```
#include <stdio.h>

#define N1 100
#define N2 9
#define N3 110

int main()
{
  float f1[N1],f2[N2],f3[N3];
  int i,j,k,n1,n2,n3;

  printf("Entre com o número de amostras\n");
  scanf("%d",&n1);
  printf("Entre com os valores das amostras\n");
  for (i=0; i < n1; i++)
    scanf("%f",&f1[i]);

  printf("Entre com o número de coeficientes do filtro\n");
  scanf("%d",&n2);
  printf("Entre com os valores dos coeficientes\n");
  for (i=0; i < n2; i++) /* ler f2 com reflexão */
    scanf("%f",&f2[n2-1-i]);

  n3 = n1+n2-1;

  /* calcula a convolução */

  for (i=0; i < n3; i++) {
    f3[i]=0.0;
    for (j=0; j < n2; j++){
      k = i+j-n2+1;
      if ((k >= 0)&&(k < n1))
        f3[i] += f1[k]*f2[j];
    }
  }

  printf("Vetor resultante\n");
  for (i=0; i < n3; i++)
    printf("%5.2f ",f3[i]);
  printf("\n");

  return 0;
}
```

7.4.3 Correlação

A **correlação** entre dois sinais discretos é definida como

$$f_3(x) = \sum_{x'=-\infty}^{x'=+\infty} f_1(x') f_2(x + x') \tag{7.3}$$

O valor máximo de $f_3(x)$ representa o deslocamento x para a esquerda que $f_2(x + x')$ tem que sofrer para minimizar a distância entre $f_1(x')$ e $f_2(x + x')$. Supondo que $f_1(x)$ e $f_2(x)$ possuem o mesmo número n de amostras (caso contrário, podemos completar com zeros o sinal com menos amostras), esses sinais podem ser alinhados usando o conceito de **correlação circular**. Isto é, podemos deslocar $f_2(x' + x)$ de um valor x para a esquerda, mas jogar as amostras com índice maior que $i > n - 1$ para o início $i\%n$ do vetor.

```
#include <stdio.h>

#define N 100

int main()
{
  float f1[N],f2[N],f3[N];
  int i,j,k,n,imax;

  printf("Entre com o número de amostras dos sinais\n");
  scanf("%d",&n);
  printf("Entre com os valores das amostras do 1o. sinal\n");
  for (i=0; i < n; i++)
    scanf("%f",&f1[i]);
  printf("Entre com os valores das amostras do 2o. sinal\n");
  for (i=0; i < n; i++)
    scanf("%f",&f2[i]);

  /* calcula a correlação circular */

  for (i=0; i < n; i++) {
    f3[i]=0.0;
    for (j=0; j < n; j++){
      f3[i] += f1[j]*f2[(j+i)%n];
    }
  }

  printf("Vetor resultante\n");
  for (i=0; i < n; i++)
    printf("%5.2f ",f3[i]);
  printf("\n");
```

```
  /* alinha o vetor f2 com f1 */

  imax = 0;
  for (i=1; i < n; i++) /* encontra o máximo */
    if (f3[i] > f3[imax])
      imax = i;

  for (i=0; i < n; i++) /* alinha f2 copiando o resultado para f3 */
    f3[i] = f2[(i+imax)%n];

  printf("Vetor alinhado\n");
  for (i=0; i < n; i++)
    printf("%5.2f ",f3[i]);
  printf("\n");

  return 0;
}
```

7.5 Exercícios

1. Refaça o programa de busca binária usando um dos algoritmos acima para ordenar o vetor.

2. Faça um programa que ler dois vetores ordenados em ordem crescente, com números n_1 e n_2 de elementos diferentes, e gera um terceiro vetor mantendo a ordem por intercalação de valores dos outros dois.

3. O histograma de um sinal discreto $f(x)$ com n amostras e L valores inteiros no intervalo $[0, L-1]$ é uma função discreta $h(l)$, $l = 0, 1, \ldots, L-1$, onde $h(l)$ é o número de ocorrências do valor $f(x) = l$, para $x = 0, 1, \ldots, n-1$. Escreva um programa para calcular o histograma de um sinal discreto.

4. Seja $p[i]$, $i = 0, 1, \ldots, n-1$, um vetor que armazena em cada posição i o coeficiente a_i de um polinômio de grau $n-1$: $a_0 + a_1 x^1 + \ldots + a_{n-1} x^{n-1}$. Faça um programa para ler um dado polinômio em p e avaliar seus valores para diferentes valores de x lidos da entrada padrão.

Capítulo 8

Matrizes

8.1 Matrizes

Vetores também podem possuir múltiplas dimensões, se declararmos um identificador que é um vetor de vetores de vetores de vetores ....

```
#define N1 10
#define N2 8
.
.
.
#define Nn 50

int main()
{
  tipo identificador[N1][N2]...[Nn]
}
```

No caso bidimensional, por exemplo, o identificador é chamado **matriz** e corresponde ao que entendemos no ensino básico por matriz (ver Figura 8.1).

```
#define NLIN 80
#define NCOL 100

int main()
{
  int m[NLIN][NCOL];
}
```

Matrizes podem ser utilizadas para cálculos envolvendo álgebra linear, para armazenar imagens, e muitas outras aplicações. O programa abaixo, por exemplo, soma duas matrizes e apresenta a matriz resultante na tela.

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

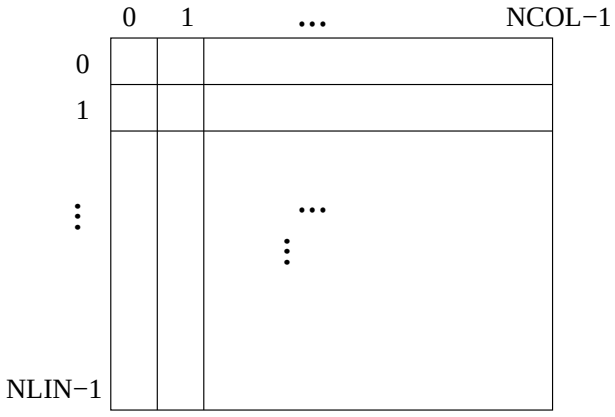

Figura 8.1: Matriz $m[NLIN][NCOL]$ de variáveis inteiras.

```
#define N 20

int main()
{
  int m1[N][N],m2[N][N],m3[N][N];
  int l,c,nlin,ncol;

  printf("Entre com os números de linhas e colunas das matrizes\n");
  scanf("%d %d",&nlin,&ncol); /* assumindo que nlin e ncol < 20 */

  printf("Entre com os elementos da matriz 1\n");
  for (l=0; l < nlin; l++)
    for (c=0; c < ncol; c++)
      scanf("%d",&m1[l][c]);

  printf("Entre com os elementos da matriz 2\n");
  for (l=0; l < nlin; l++)
    for (c=0; c < ncol; c++)
      scanf("%d",&m2[l][c]);

  /* soma as matrizes */

  for (l=0; l < nlin; l++)
    for (c=0; c < ncol; c++)
      m3[l][c] = m1[l][c] + m2[l][c];

  /* imprime o resultado */

  printf("Resultado: \n");
  for (l=0; l < nlin; l++) {
```

```
    for (c=0; c < ncol; c++)
      printf("%2d ",m3[l][c]);
    printf("\n");
  }
  return(0);
}
```

Outro exemplo é a multiplicação de matrizes.

```
#include <stdio.h>

#define N 20

int main()
{
  int m1[N][N],m2[N][N],m3[N][N];
  int l,c,i,nlin1,ncol1,nlin2,ncol2,nlin3,ncol3;

  printf("Entre com os números de linhas e colunas da matriz 1\n");
  scanf("%d %d",&nlin1,&ncol1); /* assumindo que nlin1 e ncol1 < 20 */

  printf("Entre com os elementos da matriz a\n");
  for (l=0; l < nlin1; l++)
    for (c=0; c < ncol1; c++)
      scanf("%d",&m1[l][c]);

  printf("Entre com os números de linhas e colunas da matriz 2\n");
  scanf("%d %d",&nlin2,&ncol2); /* assumindo que nlin2 e ncol2 < 20 */

  if (ncol1 != nlin2){
    printf("Erro: Número de colunas da matriz 1 está diferente\n");
    printf("      do número de linhas da matriz 2\n");
    exit(-1);
  }

  printf("Entre com os elementos da matriz 2\n");
  for (l=0; l < nlin2; l++)
    for (c=0; c < ncol2; c++)
      scanf("%d",&m2[l][c]);

  nlin3 = nlin1;
  ncol3 = ncol2;

  /* multiplica as matrizes */

  for (l=0; l < nlin3; l++)
```

```
    for (c=0; c < ncol3; c++) {
      m3[l][c] = 0;
      for (i=0; i < nlin2; i++)
        m3[l][c] = m3[l][c] + m1[l][i]*m2[i][c];
    }

  /* imprime o resultado */

  printf("Resultado: \n");
  for (l=0; l < nlin3; l++) {
    for (c=0; c < ncol3; c++)
      printf("%2d ",m3[l][c]);
    printf("\n");
  }

  return(0);
}
```

8.2 Linearização de Matrizes

Matrizes também podem ser representadas na forma unidimensional (isto é muito comum em processamento de imagens, por exemplo). Considere a matriz da figura 8.1. Podemos armazenar seus elementos da esquerda para direita e de cima para baixo iniciando em $[0,0]$ até $[NLIN-1, NCOL-1]$ em um vetor de $NLIN \times NCOL$ variáveis. Para saber o índice i do elemento do vetor correspondente a variável $m[l,c]$ da matriz, fazemos $i = l * NCOL + c$. O processo inverso é dado por $c = i\%NCOL$ e $l = i/NCOL$.

8.3 Exercícios

Consulte os livros de álgebra linear e:

1. Escreva um programa para calcular a transposta de uma matriz.
2. Escreva um programa para calcular o determinante de uma matriz.
3. Escreva um programa para inverter uma matriz.

Capítulo 9

Cadeias de caracteres

9.1 Cadeias de caracteres

Uma cadeia de caracteres (*string*) é uma seqüência de letras, símbolos (!,?,+,=,%,;, etc.), espaços em branco e/ou dígitos terminada por

```
'\0'
```

Isto é, um vetor de variáveis do tipo char.

```
#include <stdio.h>

int main()
{
  char texto[100]="O professor gost? de alunos estudiosos.";
  int i;

  texto[16]='a'; /* Corrige erro no texto */
  i=0;
  while (texto[i] != '\0'){ /* Imprime caracter por caracter
                               separados por | */
    printf("%c|",texto[i]);
    i++;
  }
  printf("\n%s\n",texto); /* Imprime o texto todo.*/

  return 0;
}
```

9.2 Lendo da entrada padrão

Até o momento vimos que o comando **scanf** pode ser usado para ler dados da entrada padrão (teclado). Mais adiante veremos que **fscanf** possui sintaxe similar, porém é mais genérico. Pois permite leitura de dados não só da entrada padrão, identificada por **stdin**, como a leitura de dados armazenados em

um **arquivo texto** no formato ASCII. A mesma observação se aplica aos comandos **printf** e **fprintf**, com identificador **stdout** para a saída padrão.

```
#include <stdio.h>

int main()
{
  char texto[100];

  fscanf(stdin,"%s",texto); /* Lê cadeias de caracteres até encontrar
                               espaço em branco, nova linha ou EOF (fim de
                               arquivo). Equivalente a scanf("%s",texto);
                               O caracter '\0' é inserido no final
                               do vetor texto após a leitura. */
return 0;
}
```

A mesma relação é válida entre outros comandos de entrada, tais como **gets** e **fgets**, **getc** (ou **getchar**) e **fgetc**, porém o comando **fgets** é mais seguro do que **gets**, pois especifica o número máximo de caracteres que o vetor suporta.

```
#include <stdio.h>

int main()
{
  char texto[100];

  fgets(texto,99,stdin); /* Lê no máximo 99 caracteres, incluindo espaços
                            em branco, até encontrar nova linha ou EOF.
                            Mais seguro que gets(texto);
                            O caracter '\0' é inserido no final
                            do vetor texto após a leitura. */
return 0;
}
```

```
#include <stdio.h>

int main()
{
  char texto[100];
  int i,c;

  i = 0;
  while (((c=fgetc(stdin))!=EOF)&&(i < 100)){/* Lê no máximo 99 caracteres ou
                                                até fim de arquivo especificado
                                                na entrada padrão com o comando
                                                < (e.g. programa < arquivo). */
```

```
    texto[i] = (char) c;
    i++;
  }
  printf("%s\n",texto); /* Imprime o texto, que pode conter várias linhas.*/

  return 0;
}
```

Uma variação deste programa seria ler apenas a primeira linha, caso usássemos '\n' em vez de EOF.

9.3 Convertendo cadeias em números, e vice-versa

Os comandos **sprintf** e **sscanf** funcionam de forma similar aos comandos printf e scanf, porém utilizando uma cadeia de caracteres no lugar da saída/entrada padrão, respectivamente.

```
#include <stdio.h>

int main()
{
  char  v1[10],v2[10],texto[100];
  float v3;
  int v4;

  sprintf(v1,"%2.1f",3.4); /* converte float para string */
  printf("%s\n",v1);

  sprintf(v2,"%2d",12);    /* converte int para string */
  printf("%s\n",v2);

  sscanf(v1,"%f",&v3);     /* converte string para float */
  printf("%2.1f\n",v3);

  sscanf(v2,"%d",&v4);     /* converte string para int */
  printf("%d\n",v4);

  sprintf(v1,"%2.1f %2d",3.4,12); /* grava números em string */
  printf("%s\n",v1);

  sscanf(v1," %f %d",&v3,&v4); /* ler números de strings. Lembre-se do bug
                                  que requer espaço em branco extra */
  printf("%2.1f %2d\n",v3,v4);

  sprintf(texto,"Os valores são %2.1f e %2d\n",v3,v4); /* gera string com
                                                          valores formatados */
  return 0;
```

```
}
```

9.4 Convertendo cadeias em números, e vice-versa (cont.)

Em algumas situações desejamos que o programa leia uma seqüência de n valores da entrada padrão. Esses valores podem ser passados, um por linha ou separados por espaços em branco (ou vírgulas).

```
#include <stdio.h>

#define N 50

int main()
{
  float v[N];
  int i,n;

  /* Opção 1 */

  fscanf(stdin,"%d",&n);
  for (i=0; i < n; i++)
    fscanf(stdin,"%f",&v[i]);

  for (i=0; i < n; i++)
    fprintf(stdout,"%f\n",v[i]);

  return 0;
}
```

Porém, em alguns casos, vamos ver que esses valores são passados em uma cadeia de caracteres. Nesses casos, o comando **strchr** pode ser usado para procurar a próxima posição da cadeia com um dado caracter (e.g. espaço em branco ou vírgula).

```
#include <stdio.h>

#define N 50

int main()
{
  char  valores[200],*vaux;
  float v[N];
  int i,n;


  fgets(valores,199,stdin);
  vaux = valores;
  sscanf(vaux,"%d",&n);
```

```
  vaux = strchr(vaux,' ') + 1;
  for (i=0; i < n; i++) {
    sscanf(vaux,"%f",&v[i]);
    vaux = strchr(vaux,' ') + 1;
  }

  for (i=0; i < n; i++)
    fprintf(stdout,"%f\n",v[i]);
  return 0;
}
```

9.5 Manipulando cadeias de caracteres

Cadeias de caracteres podem ser manipuladas para diversos fins práticos. O programa abaixo, por exemplo, ilustra algumas operações úteis.

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

int main()
{
  char s1[11], s2[11], s3[21];
  int resultado;

  fscanf(stdin,"%s",s1);
  fscanf(stdin,"%s",s2);

  /* comprimento das cadeias */

  printf("%s tem %d caracteres\n",s1,strlen(s1));
  printf("%s tem %d caracteres\n",s2,strlen(s2));

  resultado = strcmp(s1,s2); /* compara strings */

  if (resultado == 0)
    printf("%s = %s\n",s1,s2);
  else
    if (resultado < 0){
      printf("%s < %s\n",s1,s2);
      strcat(s3,s1); /* concatena strings */
      strcat(s3,s2);
      printf("%s\n",s3);
    }else{
      printf("%s > %s\n",s1,s2);
      strcat(s3,s2);
      strcat(s3,s1);
```

```
      printf("%s\n",s3);
    }

return 0;
}
```

Veja também as funções **strcpy** e **strncpy** para realizar cópias de caracteres e **strncat** para concatenação de n caracteres no final de uma cadeia.

Um exemplo prático é a busca de padrões em uma cadeia de caracteres. Este problema é muito comum quando pretendemos localizar palavras em um texto e padrões em uma seqüência de caracteres, como ocorre em Bioinformática. O algoritmo abaixo ilustra uma solução $O(nm)$, onde n é o comprimento da cadeia e m é o comprimento do padrão.

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

int main()
{
  char cadeia[101], padrao[11];
  int i,n,m;

  fscanf(stdin,"%s",padrao);
  fscanf(stdin,"%s",cadeia);

  n  = strlen(cadeia);
  m  = strlen(padrao);

  i = 0;
  while (i <= (n-m)){
    /* compara m caracteres */
    if (strncmp(&cadeia[i],padrao,m)==0)
      printf("Padrão encontrado na posição %d\n",i);
    i++;
  }

return 0;
}
```

9.6 Exercícios

Quando o padrão não é encontrado a partir da posição i, o algoritmo acima repete a busca a partir da posição $i+1$. Porém, quando os k primeiros caracteres são iguais aos do padrão e a diferença ocorre no caracter da posição $i+k$, a nova busca pode iniciar nesta posição. Note também que toda vez que o padrão for encontrado na posição i, a próxima busca pode iniciar na posição $i+m$. Com essas observações, escreva um algoritmo mais eficiente para buscar padrões.

Capítulo 10

Registros

Sabemos que variáveis compostas são aquelas que agrupam um certo número de elementos em um único identificador. No caso de vetores e matrizes, todos os elementos agrupados são do mesmo tipo e, portanto, dizemos que são variáveis compostas homogêneas.

Em muitas situações, porém, desejamos agrupar variáveis de tipos diferentes. Um exemplo é o caso em que agrupamos dados sobre uma pessoa/objeto (e.g. RA, nome, e notas de um aluno). O conceito de registro permite criar um identificador único associado a todos esses dados.

10.1 Registros

Um **registro** é uma variável composta heterogênea e seus elementos são chamados **campos**.

Diferentes tipos de registro podem ser criados com campos diferentes. Para definir um tipo de registro, nós usamos o comando **typedef struct**. O programa abaixo define um tipo de registro, Aluno, uma variável deste tipo, armazena dados nos seus campos e depois imprime os dados armazenados.

```
#include <stdio.h>

typedef struct _aluno {
  int   RA;
  char  nome[50];
  float nota[3];
} Aluno;

int main()
{
  Aluno a; /* variável do tipo Aluno */

  a.RA      = 909090;
  a.nome    = "Jose Maria";
  a.nota[0] = 3.0;
  a.nota[1] = 10.0;
  a.nota[2] = 7.0;
  printf("%6d %s %5.2f %5.2f %5.2f\n",a.RA,a.nome,a.nota[0],a.nota[1],a.nota[2]);
```

```
  return 0;
}
```

Observe que cada campo do registro é uma variável de qualquer tipo válido, incluindo um **outro registro, vetor, matriz**, etc.

```
#include <stdio.h>

typedef struct _ponto {
  float x;
  float y;
} Ponto;

typedef struct _reta {
  Ponto p1;
  Ponto p2;
} Reta;

typedef struct _curva { /* pontos consecutivos são interligados
                           por segmentos de reta. */
  Ponto pt[100];
  int npts;
} Curva;

int main()
{
  Reta r;
  Curva c;
  int i;

  /* ler os pontos da reta */

  scanf("%f %f",&r.p1.x,&r.p1.y);
  scanf("%f %f",&r.p2.x,&r.p2.y);

  /* ler os pontos da curva */

  scanf("%d",&c.npts);
  for (i=0; i < c.npts; i++)
    scanf("%f %f",&c.pt[i].x,&c.pt[i].y);

  /* complete o programa para que ele verifique se existe
     intersecção entre a curva e a reta. */

  return 0;
}
```

Vetores de registros podem ser usados para armazenar base de dados (ou parte da base) em memória. O programa abaixo ilustra o armazenamento de uma mini base com 5 nomes e 5 telefones.

```
#include <stdio.h>

typedef struct _agenda {
  char nome[50];
  int  telefone;
} Agenda;

int main()
{
  Agenda amigo[5];
  char nome_aux[50];
  int i,comp;

  printf("Entre com os nomes\n");
  for (i=0; i < 100; i++) {
    fgets(nome_aux,49,stdin);
    comp = strlen(nome_aux);
    strncpy(amigo[i].nome,nome_aux,comp-1); /* elimina \n */
    amigo[i].nome[comp-1] = '\0'; /* insere \0 por garantia */
  }

  printf("Entre com os telefones\n");
  for (i=0; i < 5; i++)
    fscanf(stdin,"%d",&amigo[i].telefone);

  for (i=0; i < 5; i++)
    fprintf(stdout,"%s: %d\n",amigo[i].nome,amigo[i].telefone);

  return 0;
}
```

10.2 Exercícios

1. Complete o programa acima para cálculo de intersecções entre a reta e a curva.

2. O centro de gravidade (x_g, y_g) de um conjunto de pontos (x_i, y_i), $i = 0, 1, \ldots, n-1$, é definido por:

$$\begin{aligned} x_g &= \frac{\sum_{i=0}^{i=n-1} x_i}{n} \\ y_g &= \frac{\sum_{i=0}^{i=n-1} y_i}{n}. \end{aligned}$$

   Faça um programa para ler os pontos de uma curva e calcular seu centro de gravidade.

3. Faça um programa para armazenar 20 nomes e 20 telefones em um vetor de registros, colocar os elementos do vetor em ordem crescente de nome, e depois imprimir o vetor ordenado.

Capítulo 11

Funções

11.1 Funções

Vimos que diversos comandos (e.g. scanf(), printf(), strlen(), strcmp(), sin(), cos(), etc) realizam tarefas mais complexas sobre valores de entrada. Esses comandos são denominados **funções**, pois consistem de agrupamentos de instruções (i.e. seqüências de comandos com uma determinada finalidade) assim como a função principal, **main()**, que agrupa as instruções do programa.

Uma **função**, portanto, é uma seqüência de comandos que pode ser executada a partir da função principal (ou de qualquer outra função).

As funções simplificam a codificação e permitem uma melhor estruturação do programa, evitando que uma mesma seqüência de comandos seja escrita diversas vezes no corpo (**escopo**) da função principal. Por exemplo, suponha um programa para calcular o número $C(n,p) = \frac{n!}{p!(n-p)!}$ de combinações de n eventos em conjuntos de p eventos, $p \leq n$. Sem o conceito de função, teríamos que repetir três vezes as instruções para cálculo do fatorial de um número x. Com o conceito de função, precisamos apenas escrever essas instruções uma única vez e substituir x por n, p, e $(n-p)$ para saber o resultado de cada cálculo fatorial.

```
#include <stdio.h>

double fatorial(int x); /* protótipo da função */

/* escopo da função */

double fatorial(int x)
{
  double fat=1;
  int i;

  for (i=x; i > 1; i--)
    fat = fat * i;

  return(fat);
}
```

```
/* função principal */

int main()
{
  int n,p,C;

  scanf("%d %d",&n,&p);
  if ((p >= 0)&&(n >= 0)&&(p <= n)){ /* chamada da função */
    C = (int)(fatorial(n)/(fatorial(p)*(fatorial(n-p))));
    printf("%d \n",C);
  }
  return 0;
}
```

A função **fatorial** recebe o valor de uma variável da função principal, armazena em uma variável x do seu escopo, e retorna o cálculo do fatorial de x para o programa principal. As variáveis x, i, e fat declaradas na função fatorial são denominadas **variáveis locais** desta função. Elas só existem na memória enquanto a função fatorial estiver sendo executada (no exemplo, elas são alocadas e desalocadas três vezes na memória), e só podem ser usadas no escopo desta função. A mesma observação é válida com relação às variáveis locais n, p, e C da função principal, porém essas permanecem na memória durante toda a execução do programa.

Um programa pode ter várias funções e uma função pode conter chamadas a outras funções do programa. Cada função deve ser declarada da seguinte forma:

```
tipo funcao(tipo var1, tipo var2,..., tipo varn); /* protótipo */


tipo funcao(tipo var1, tipo var2,..., tipo varn)
{
  /* escopo */

  return (valor);
}
```

O tipo do valor retornado pela função deve ser compatível com o tipo da função. O tipo **void** é usado quando a função não retorna valor. Os valores de entrada e saída de uma função são denominados **parâmetros**. Esses parâmetros podem ser de qualquer tipo válido, incluindo registros, apontadores, cadeias de caracteres, vetores, etc.

11.2 Parâmetros passados por valor e por referência

No caso da função fatorial, o valor de n na chamada fatorial(n) é passado para uma cópia x da variável n. Qualquer alteração em x não afeta o conteúdo de n no escopo da função principal. Dizemos então que o parâmetro é passado por **valor**. Isto nos permite, por exemplo, simplificar a função para:

```
double fatorial(int x)
{
  double fat=1;

  while (x > 1){
    fat = fat * x;
    x--;
  }

  return(fat);
}
```

Porém, em várias situações desejamos alterar o conteúdo de uma ou mais variáveis no escopo da função principal. Neste caso, os parâmetros devem ser passados por **referência**. Isto é, a função cria uma cópia do endereço da variável correspondente na função principal em vez de uma cópia do seu conteúdo. Qualquer alteração no conteúdo deste endereço é uma alteração direta no conteúdo da variável da função principal. Por exemplo, o programa acima requer que $p \leq n$. Caso contrário, podemos trocar o conteúdo dessas variáveis.

```
#include <stdio.h>

double  fatorial(int x);
void troca(int *x, int *y); /* x e y são apontadores para endereços de
                               memória que guardam valores do tipo int */

double fatorial(int x)
{
  double fat=1;
  while (x > 1){
    fat = fat * x;
    x--;
  }
  return(fat);
}

void troca(int *x, int *y)
{
  int aux;

  aux = *x;  /* conteúdo de x é atribuído ao conteúdo de aux */
  *x  = *y;  /* conteúdo de y é atribuído ao conteúdo de x */
  *y  = aux; /* conteúdo de aux é atribuído ao conteúdo de y */
}

int main()
{
```

```
  int n,p,C;

  scanf("%d %d",&n,&p);
  if (p > n)
    troca(&p,&n); /* passa os endereços de p e de n */

  if ((p >= 0)&&(n >= 0)){
    C = (int)(fatorial(n)/(fatorial(p)*(fatorial(n-p))));
    printf("%d \n",C);
  }
  return 0;
}
```

11.3 Hierarquia entre funções

Como mencionado na aula anterior, uma função pode conter chamadas a outras funções. Após a execução de uma função, o programa sempre retorna para a posição imediatamente após a sua chamada. O exemplo abaixo demonstra leitura, ordenação e busca binária em uma lista de telefones de amigos.

```
#include <stdio.h>

#define N 5 /* tamanho da agenda de amigos, que na prática deve ser bem maior
               para justificar a busca binária. */

typedef enum {false,true} bool;

typedef struct _agenda {
  char nome[50];
  int  telefone;
} Agenda;

void      le_nome(char *nome); /* lê nome trocando \n por \0 */

void      le_dados(Agenda *amigo); /* lê nome e telefone de cada
                                      amigo, carregando esses dados
                                      em um vetor de registros. */
void      troca(Agenda *t1, Agenda *t2); /* troca dados do
                                            conteúdo de dois
                                            registros */
void      ordena(Agenda *amigo); /* ordena registros por seleção */
/* Realiza buscas binárias por nome, retornando false se o nome não
   for encontrado, e true no caso contrário. Neste caso, retorna o
   telefone encontrado no endereço apontado pela variável telefone. */
bool      busca(Agenda *amigo, char *nome, int *telefone);
```

```
/* Escopos das funções */

void le_nome(char *nome)
{
  int i,comp;

  fgets(nome,49,stdin);
  comp = strlen(nome);
  nome[comp-1]='\0'; /* sobrescreve \0 no lugar de \n */
}

void le_dados(Agenda *amigo)
{
  char linha[100],*p;
  int  i,comp;

  for (i=0; i < N; i++) {
    fgets(linha,99,stdin); /* lê linha */
    p = (char *)strchr(linha,'#'); /* procura posição do delimitador */
    comp = p-linha-1; /* acha o número de caracteres antes do delimitador */
    strncpy(amigo[i].nome,linha,comp); /* copia esses caracteres */
    amigo[i].nome[comp]='\0'; /* acrescenta o final de string */
    sscanf(p+1,"%d",&amigo[i].telefone); /* lê telefone a partir da
                                            posição do delimitador + 1 */
  }
}

void troca(Agenda *t1, Agenda *t2)
{
  Agenda t;

  t   = *t1;
  *t1 = *t2;
  *t2 = t;
}

void ordena(Agenda *amigo)
{
  int i,j,jm;

  for (j=0; j < N-1; j++){
  /* seleciona o menor nome */
    jm    = j;
    for (i=j+1; i < N; i++){
      if (strcmp(amigo[i].nome,amigo[jm].nome) < 0){
        jm    = i;
```

```
      }
    }

    /* troca */
    if (j != jm)
      troca(&amigo[j],&amigo[jm]);
  }
}

bool busca(Agenda *amigo, char *nome, int *telefone)
{
  int inicio, fim, pos;
  bool achou;


    inicio = 0;
    fim    = N-1;
    achou  = false;

    while ((inicio <= fim)&&(!achou)){
      pos = (inicio+fim)/2;
      if (strcmp(nome,amigo[pos].nome) < 0)
        fim = pos-1;
      else
        if (strcmp(nome,amigo[pos].nome) > 0)
          inicio = pos + 1;
        else{
          achou = true;
          *telefone = amigo[pos].telefone;
        }
    }
    return(achou);
}

int main()
{
  Agenda amigo[N];
  char nome[50];
  int telefone;

  le_dados(amigo);
  ordena(amigo);

  printf("Entre com um nome ou \nfim para sair do programa:\n");
  le_nome(nome);
```

```
  while(strcmp(nome,"fim")){
    if (busca(amigo,nome,&telefone))
      printf("O número do telefone de %s é: %d\n",nome,telefone);
    else
      printf("%s não está na lista de amigos\n",nome);

    printf("Entre com um nome ou\nfim para sair do programa\n");
    le_nome(nome);
  }
  return 0;
}
```

11.4 Exercícios

Reescreva programas das aulas anteriores utilizando funções.

Capítulo 12

Recursão

12.1 Soluções recursivas

A solução de um problema é dita **recursiva** quando ela é escrita em função de si própria para instâncias menores do problema. Considere, por exemplo, o problema de calcular a soma dos números inteiros no intervalo $[m, n]$, onde $m, n \in \mathbb{Z}$ e $m \leq n$.

A solução **iterativa** é $m + (m+1) + (m+2) + \ldots + n$, que pode ser facilmente implementada com a função abaixo:

```
int Soma(int m, int n)
{
  int soma=0, i;

  for (i=m; i <= n; i++)
    soma += i;

  return(soma);
}
```

A solução recursiva está diretamente ligada ao conceito de indução matemática. A **indução fraca** usa como hipótese que a solução de um problema de tamanho t pode ser obtida a partir de sua solução de tamanho $t-1$. Por exemplo, se soubermos o resultado da soma de m até $n-1$, basta somar n a este resultado: $Soma(m, n) = Soma(m, n-1) + n$. Neste caso, dizemos que a recursão é **decrescente**. Podemos também obter o mesmo resultado somando m com a soma de $m+1$ até n: $Soma(m, n) = m + Soma(m+1, n)$. Neste caso, dizemos que a recursão é **crescente**. A solução recursiva, portanto, pode ser implementada com funções do tipo:

```
tipo função(<parâmetros>)
{
  <variáveis locais>

  if (<condição de parada>)
  {
    <comandos finais>
    return (valor);
```

```
  } else {
    <comandos iniciais>
    <chamada recursiva>
    <comandos finais>
    return (valor);
  }
}
```

No caso do exemplo acima temos:

- Solução crescente

```
int Soma(int m, int n)
{
  if (m==n){
    return(n);
  } else {
    return(m+Soma(m+1,n));
  }
}
```

- Solução decrescente

```
int Soma(int m, int n)
{
  if (m==n){
    return(m);
  } else {
    return(Soma(m,n-1)+n);
  }
}
```

Observe que nestes exemplos não precisamos de variáveis locais, nem de comandos iniciais e finais, portanto podemos colocar a chamada recursiva no comando de retorno. As **árvores de recursão** mostradas na Figura 12.1 ajudam a visualizar o comportamento dessas funções.

A **indução forte** usa outra hipótese: a solução de um problema de tamanho t depende de suas soluções de tamanhos t', para todo $t' < t$. Esta estratégia também é denominada **divisão e conquista**. Por exemplo, podemos dizer que $Soma(m, n) = Soma(m, \frac{m+n}{2}) + Soma(\frac{m+n}{2} + 1, n)$. Ou seja, o problema é dividido em subproblemas similares (divisão), os subproblemas são resolvidos recursivamente (i.e. quando o problema chega a seu tamanho mínimo, ele é resolvido de forma direta), e as soluções dos subproblemas são combinadas para gerar a solução do problema de tamanho maior (conquista). Isto requer, porém, ao menos duas chamadas recursivas:

```
tipo função(<parâmetros>)
{
  <variáveis locais>
```

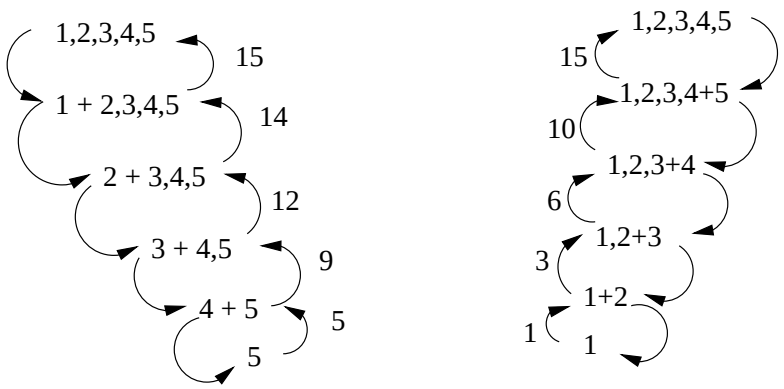

Figura 12.1: Árvores de recursão para $m = 1$ e $n = 5$.

```
  if (<condição de parada>)
  {
    <comandos finais>
    return (valor);
  } else {
    <comandos iniciais>
    <primeira chamada recursiva>
    <segunda  chamada recursiva>
    <comandos finais>
    return (valor);
  }
}
```

No caso do exemplo acima temos:

```
int Soma(int m, int n)
{
  if (m==n){
    return(m);
  } else {
    return(Soma(m,(m+n)/2)+Soma((m+n)/2+1,n));
  }
}
```

Mais uma vez, pelas mesmas razões acima, as chamadas recursivas podem ser feitas no comando de retorno. A árvore de recursão é apresentada na Figura 12.2.

Observe que em todos os exemplos, as chamadas recursivas são para a própria função. Dizemos então que a **recursão é direta**. Em alguns problemas (e.g. análise sintática do compilador), porém, uma função A pode chamar uma função B que chama C, que chama ..., que chama A. Neste caso dizemos que a **recursão é indireta**.

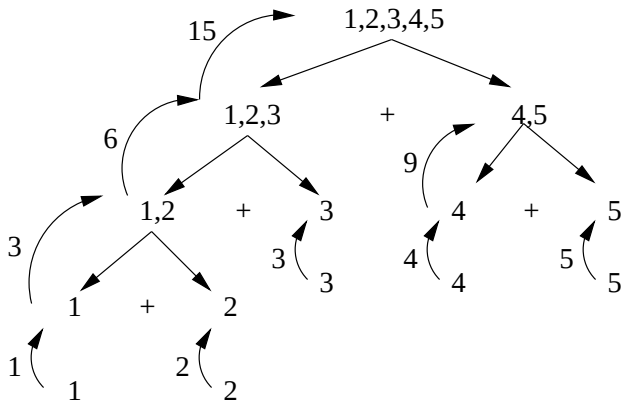

Figura 12.2: Árvore de recursão para $m = 1$ e $n = 5$.

12.2 Ordenação por indução fraca

Os algoritmos de ordenação de seqüências por seleção, inserção, e permutação podem ser implementados usando indução fraca. Na ordenação por inserção, ordenamos a seqüência até a penúltima posição e inserimos o último elemento na posição correta da seqüência ordenada. Na ordenação por seleção, selecionamos o maior elemento, colocamos ele na última posição e depois repetimos o processo para a subseqüência terminada no penúltimo. Na ordenação por permutação, o processo é similar. Os elementos são permutados até que o maior seja o último, e depois repetimos o processo para a subseqüência terminada no penúltimo.

Por exemplo, a função abaixo ordena de forma recursiva um vetor v de inteiros e de tamanho n por inserção.

```
void Insercao(int *v, int n)
{
  int i,j; /* variáveis locais */

  /* A condição de parada é não fazer nada. Caso contrário: */
  if (n > 1) {
    /* comandos iniciais: vazio */
    /* chamada recursiva */
    Insercao(v,n-1);
    /* comandos finais: insere o último elemento na posição correta. */
    i = n-2; j=n-1;
    while ( (i >= 0) && (v[i] > v[j])){
      troca(&v[i],&v[j]);
      i--; j--;
    }
  }
}
```

12.3 Ordenação por indução forte

A vantagem da indução forte é reduzir a complexidade da ordenação de $O(n^2)$ para $O(n \log n)$. O algoritmo mais simples nesta linha é o $merge - sort$. Este algoritmo subdivide a seqüência em duas, ordena de forma recursiva cada parte, e depois intercala as partes ordenadas.

```
/* Considerando que o vetor está ordenado do início até o meio e do
   meio + 1 até o final, intercala seus elementos para que fique
   ordenado do início ao fim. */

void Intercala(int *v, int inicio, int meio, int fim)
{
  int i,j,k,vaux[N]; /* Ordenação requer memória auxiliar do mesmo
                        tamanho da entrada. */

  i=inicio;
  j=meio+1;
  k=inicio;

  while((i<=meio)&&(j<=fim)){
    if (v[i] <= v[j]){
      vaux[k]=v[i];
      i++; k++;
    }else{
      vaux[k]=v[j];
      j++; k++;
    }
  }
  for(i=i; i <= meio; i++,k++)
    vaux[k]=v[i];
  for(j=j; j <= fim; j++,k++)
    vaux[k]=v[j];
  /* copia de volta para v */
  for (i=inicio; i <= fim; i++)
    v[i]=vaux[i];
}

void MergeSort(int *v, int inicio, int fim)
{
  int meio; /* variável local */

  /* A condição de parada é não fazer nada. Caso contrário: */
  if (inicio < fim) {
    /* comando inicial: calcula o meio */
    meio = (inicio+fim)/2;
    /* chamadas recursivas */
```

```
    MergeSort(v,inicio,meio);
    MergeSort(v,meio+1,fim);
    /* comando final: intercalação */
    Intercala(v,inicio,meio,fim);
  }
}
```

Uma desvantagem do algoritmo acima, porém, é a necessidade de memória auxiliar, na função de intercalação, do mesmo tamanho da entrada. Isto pode ser um problema para seqüências muito grandes.

Outro algoritmo que usa indução forte, tem complexidade $O(n \log n)$ no caso médio, e $O(n^2)$ no pior caso, mas não requer memória auxiliar é o $quick - sort$. Este algoritmo particiona a seqüência em duas partes de tal forma que todos os elementos da primeira parte são menores ou iguais aos da segunda. A seqüência é ordenada repetindo-se este processo de forma recursiva para cada parte.

```
void QuickSort(int *v, int inicio, int fim)
{
  int p;

  if (inicio < fim){
    p = Particiona(v,inicio,fim);
    QuickSort(v,inicio,p);
    QuickSort(v,p+1,fim);
  }
}
```

12.4 Exercícios

1. Escreva uma função recursiva para calcular o fatorial de um número inteiro n. Note que o fatorial de n pode ser definido de forma recursiva como:

$$fat(n) \begin{cases} 1 & \text{se } n = 0 \\ n \times fat(n-1) & \text{se } n > 0 \end{cases} \tag{12.1}$$

2. Os termos de uma seqüência de Fibonacci (e.g. $\langle 0, 1, 1, 2, 3, 5, 8, \ldots \rangle$ são definidos de forma recursiva como:

$$fib(n) = \begin{cases} n - 1 & \text{se } 1 \leq n \leq 2 \\ fib(n-1) + fib(n-2) & \text{se } n > 2 \end{cases} \tag{12.2}$$

   Escreva uma função recursiva para retornar o n-ésimo termo desta seqüência.

3. Escreva as funções de ordenação de forma recursiva, por seleção e por permutação, de um vetor v com n elementos.

4. Escreva uma função recursiva para ordenar v por partição. Isto é, complete o código do $quick - sort$.

Capítulo 13

Alocação dinâmica de memória

13.1 Declarando e manipulando apontadores

Antes de falarmos de alocação dinâmica de memória, vamos entender como variáveis do tipo apontador são definidas e como elas podem ser manipuladas em memória.

```
int main()
{
  int v[10],*u;

  v[0] = 30;
  printf("%d\n",*v); /* imprime o conteúdo apontado por v, que é v[0] */
  u = v; /* atribui o endereço do vetor a u */
  printf("%d\n",u);
  u[4] = 10; /* atribui 10 a u[4], que é v[4] */
  printf("%d\n",v[4]);
  u = &v[2]; /* atribui o endereço de v[2] a u */
  printf("%d\n",u);
  u[2] = 20; /* atribui 20 a u[2], que é v[4] */
  printf("%d\n",v[4]);
}
```

A memória disponível para um programa é dividida em três partes. Uma parte é reservada para o código executável do programa, e as outras duas são reservadas para variáveis definidas durante a execução do programa. Por exemplo, o programa acima utiliza o espaço disponível na **pilha**— parte da memória reservada para variáveis locais, incluindo as variáveis associadas a funções— para armazenar as variáveis v, u, v[0], v[1], ..., v[9]. Este espaço é normalmente insuficiente para armazenar muitas variáveis, como é o caso de vetores muito grandes. Uma outra situação ocorre quando não sabemos *a priori* o tamanho do vetor. Para esses casos, existe uma parte para **dados** na memória, bem maior que a pilha, a qual pode ser alocada pelo programador. Nesta aula, vamos aprender como definir e manipular esses espaços em memória.

13.2 Alocando memória para vetores

O programa abaixo mostra como alocar e desalocar espaço em memória durante a sua execução.

```
#include <stdio.h>
#include <malloc.h> /* biblioteca de funções de alocação de memória */

typedef struct _ponto {
  float x,y;
} Ponto;

int main()
{
  Ponto *p=NULL; /* variável do tipo apontador para Ponto */
  int *v=NULL; /* variável do tipo apontador para int */
  char *text=NULL; /* variável do tipo apontador para char */
  int n;

  scanf("%d",&n);
  v = (int *)calloc(n,sizeof(int)); /* aloca n elementos inteiros em memória
                                       e retorna o endereço do primeiro, ou
                                       NULL caso não tenha sido alocada. */
  scanf("%d",&n);
  text = (char *)calloc(n,sizeof(char)); /* aloca espaço para cadeia
                                            com n caracteres. */
  scanf("%d",&n);
  p = (Ponto *)calloc(n,sizeof(Ponto)); /* aloca espaço para vetor de
   pontos com n pontos. */

  printf("v %d text %d p %d\n",v,text,p); /* imprime os endereços */

  v[0] = 10;
  sprintf(text,"Oi"); /* text = "Oi" está errado neste caso, pois o
  computador alocará novo espaço para armazenar a cadeia e atribuirá o
  endereço deste espaço para text. Portanto, o endereço alocado
  inicialmente para text ficará perdido. Assim, devemos usar o comando
  sprintf para gravar a cadeia no espaço alocado para text. */

  p[0].x = 20;
  p[0].y = 30;
  printf("%d\n",v[0]);
  printf("%s\n",text);
  printf("(%f,%f)\n",p[0].x,p[0].y);

  if (v != NULL) {
    free(v); /* libera memória */
```

```
    v = NULL;
  }

  if (text != NULL) {
    free(text); /* libera memória */
    text = NULL;
  }

  if (p != NULL) {
    free(p); /* libera memória */
    p = NULL;
  }
return 0;
}
```

Outra opção para alocação de memória é usar a função *malloc*:

```
v = (int *) malloc(n*sizeof(int));      /* só possui 1 argumento */
text = (char *) malloc(n*sizeof(char)); /* só possui 1 argumento */
p = (Ponto *) malloc(n*sizeof(Ponto));  /* só possui 1 argumento */
```

A única diferença entre *malloc* e *calloc*, fora o número de argumentos, é que *calloc* inicializa a memória com zeros/espaços em branco.

Portanto, podemos usar a sintaxe abaixo para alocar espaço para vetores de todos os tipos conhecidos.

```
tipo *var=(tipo *) calloc(número de elementos,sizeof(tipo));
tipo *var=(tipo *) malloc(número de elementos*sizeof(tipo));
```

13.3 Alocando memória para estruturas abstratas

Por uma questão de organização, quando usamos registro para definir uma estrutura abstrata, nós devemos definir uma função para cada tipo de operação com esta estrutura. Por exemplo, suponha uma estrutura para representar uma figura formada por um certo número de pontos interligados por segmentos de reta. As operações desejadas são alocar memória para a figura, ler os pontos da figura, imprimir os pontos lidos, e desalocar memória para a figura. Estas operações definem, portanto, quatro funções conforme o programa abaixo.

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

typedef struct _ponto {
  float x,y; /* coordenadas Cartesianas de um ponto */
} Ponto;

typedef struct _figura {
  Ponto *p; /* pontos da figura */
```

```
  int n;    /* número de pontos */
} Figura;

Figura CriaFigura(int n); /* aloca espaço para figura com n pontos e
                             retorna cópia do registro da figura */
void   DestroiFigura(Figura f); /* desaloca espaço */
Figura LeFigura(); /* lê o número de pontos, cria a figura, ler os
                      pontos, e retorna cópia do registro da
                      figura. */
void   ImprimeFigura(Figura f); /* imprime os pontos */

Figura CriaFigura(int n)
{
  Figura f;

  f.n = n;
  f.p = (Ponto *) calloc(f.n,sizeof(Ponto));
  return(f); /* retorna cópia do conteúdo de f */
}

void DestroiFigura(Figura f)
{
  if (f.p != NULL) {
    free(f.p);
  }
}

Figura LeFigura()
{
  int i,n;
  Figura f;

  scanf("%d",&n); /* lê o número de pontos */
  f = CriaFigura(n); /* aloca espaço */
  for (i=0; i < f.n; i++) /* lê os pontos */
    scanf("%f %f",&(f.p[i].x),&(f.p[i].y));
  return(f); /* retorna cópia do conteúdo de f */
}

void ImprimeFigura(Figura f)
{
  int i;

  for (i=0; i < f.n; i++)
    printf("(%f,%f)\n",f.p[i].x,f.p[i].y);
}
```

```
int main()
{
  Figura f;

  f=LeFigura();
  ImprimeFigura(f);
  DestroiFigura(f);
  return(0);
}
```

Um defeito do programa acima é que todos os campos do registro f são armazenados na pilha. No caso de registros com vários campos, estamos disperdiçando espaço na pilha. Podemos evitar isto se definirmos uma **variável apontador para o tipo Figura**, em vez de uma variável do tipo Figura. Esta mudança requer as seguintes alterações no programa.

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

typedef struct _ponto {
  float x,y; /* coordenadas Cartesianas de um ponto */
} Ponto;

typedef struct _figura {
  Ponto *p; /* pontos da figura */
  int n;    /* número de pontos */
} Figura;

Figura *CriaFigura(int n); /* Aloca espaço para figura com n pontos e
      retorna endereço do registro da figura */
void   DestroiFigura(Figura **f); /* Destrói espaço alocado e atribui
                                     NULL para a variável apontador
                                     f. Paratanto, precisamos passar o
                                     endereço do apontador, que é uma
                                     variável do tipo apontador de
                                     apontador, ou melhor, apontador
                                     duplo. */
Figura *LeFigura(); /* Lê o número de pontos, cria figura, lê os
                       pontos, e retorna o endereço do registro da
                       figura */
void   ImprimeFigura(Figura *f); /* Imprime os pontos */

Figura *CriaFigura(int n)
{
  Figura *f=(Figura *)calloc(1,sizeof(Figura)); /* aloca espaço para 1
                                                   registro do tipo
```

```
                                                Figura e retorna
                                                seu endereço em f */


  f->p = (Ponto *) calloc(n,sizeof(Ponto)); /* aloca espaço para vetor
                                               de pontos e retorna seu
                                               endereço no campo f->p
                                               do registro apontado
                                               por f. Note a mudança
                                               de notação de f.p para
                                               f->p, pois f agora é
                                               apontador de registro e
                                               não mais registro */

  f->n = n; /* copia o número de pontos para o campo f->n */

  return(f); /* retorna endereço do registro alocado em memória */
}

void DestroiFigura(Figura **f)
{
  if (*f != NULL) { /* *f representa o conteúdo do apontador duplo,
                       que é o endereço armazenado no apontador f. */
    if ((*f)->p != NULL)
      free((*f)->p);
    *f = NULL; /* atribui NULL para a variável do escopo principal. */
  }
}

Figura *LeFigura()
{
  int i,n;
  Figura *f=NULL;

  scanf("%d",&n);
  f = CriaFigura(n);
  for (i=0; i < f->n; i++)
    scanf("%f %f",&(f->p[i].x),&(f->p[i].y));
  return(f); /* retorna endereço do registro alocado em memória */
}

void ImprimeFigura(Figura *f)
{
  int i;

  for (i=0; i < f->n; i++)
```

```
    printf("(%f,%f)\n",f->p[i].x,f->p[i].y);
}

int main()
{
  Figura *f=NULL; /* variável do tipo apontador para registro */

  f=LeFigura();
  ImprimeFigura(f);
  DestroiFigura(&f); /* passa f por referência, i.e. o endereço de f. */
  return(0);
}
```

13.4 Alocando memória para matrizes

No caso de vetores multidimensionais (e.g. matrizes), devemos alocar um vetor de apontadores, i.e. um apontador por linha, e depois um vetor de elementos para cada linha. Assim como fizemos para Figura, podemos entender uma matriz como uma estrutura abstrata com operações de criação, destruição, transposição, leitura e impressão. O programa abaixo ilustra esses conceitos para o caso de matriz representada por um registro. Fica como exercício modificá-lo para representar a matriz como apontador para registro, assim como fizemos na aula de alocação de memória para vetores.

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

typedef struct _matriz {
  float **elem; /* matriz de elementos do tipo float */
  int nlin,ncol; /* números de linhas e colunas */
} Matriz;

Matriz CriaMatriz(int nlin, int ncol);
Matriz LeMatriz();
Matriz TranspoeMatriz(Matriz m1);
void   ImprimeMatriz(Matriz m);
void   DestroiMatriz(Matriz m);

Matriz CriaMatriz(int nlin, int ncol)
{
  Matriz m;
  int l;

  m.elem = (float **)calloc(nlin,sizeof(float *)); /* aloca vetor de
                                                      apontadores para
                                                      variáveis do
                                                      tipo float */
  if (m.elem != NULL){
```

```
    for (l=0; l < nlin; l++)
      m.elem[l] = (float *)calloc(ncol,sizeof(float)); /* aloca vetor
                                                          de variáveis
                                                          do tipo
                                                          float */

    m.nlin = nlin;
    m.ncol = ncol;
  }

  return(m); /* retorna cópia do registro */
}

Matriz  LeMatriz()
{
  int l,c,nlin,ncol;
  Matriz m;

  scanf("%d %d",&nlin,&ncol);
  m = CriaMatriz(nlin,ncol);
  for (l=0; l < m.nlin; l++)
    for (c=0; c < m.ncol; c++)
      scanf("%f",&m.elem[l][c]);
  return(m);
}

Matriz TranspoeMatriz(Matriz m1)
{
  int l,c;
  Matriz m2;

  m2 = CriaMatriz(m1.ncol,m1.nlin);
  for (l=0; l < m2.nlin; l++)
    for (c=0; c < m2.ncol; c++)
      m2.elem[l][c] = m1.elem[c][l];
  return(m2);
}

void   ImprimeMatriz(Matriz m)
{
  int l,c;

  for (l=0; l < m.nlin; l++){
    for (c=0; c < m.ncol; c++)
      printf("%f ",m.elem[l][c]);
    printf("\n");
```

```
  }
}

void DestroiMatriz(Matriz m)
{
  int l;

  if (m.elem != NULL) {
    for (l=0; l < m.nlin; l++) /* desaloca espaço do vetor de
                                  variáveis de cada linha */
      if (m.elem[l] != NULL)
        free(m.elem[l]);
    free(m.elem); /* desaloca espaço do vetor de apontadores */
  }
}

int main()
{
  Matriz m1,m2;

  m1 = LeMatriz();
  m2 = TranspoeMatriz(m1);
  ImprimeMatriz(m2);
  DestroiMatriz(m1);
  DestroiMatriz(m2);

  return 0;
}
```

13.5 Outras formas de manipulação de apontadores duplos

O programa abaixo ilustra outras formas de manipulação de apontadores duplos, que costumam confundir bastante os programadores.

```
#include <malloc.h>

/*----- 1. Manipulação com alocação e desalocação dinâmicas de memória ----*/

float **CriaMatriz(int nlin, int ncol); /* Aloca memória para matriz
                                           representada por apontador
                                           duplo */

void    DestroiMatriz(float ***m, int nlin); /* Desaloca memória
                                                atribuindo NULL ao
                                                conteúdo de m. Isto
                                                requer passar o
```

```
                                                endereço de m para a
                                                função, pontanto m é um
                                                apontador de apontador
                                                duplo. Ou seja,
                                                apontador triplo. */

/*----- 2. Manipulação no caso de alocação estática ----------------------*/

void TraspoeMatriz(float m1[2][3], float m2[3][2]); /* Transpõe m1 para m2 */

/*---------------------- Escopos das funções -----------------------------*/

float **CriaMatriz(int nlin, int ncol)
{
  float **m=NULL;
  int l;

  m = (float **)calloc(nlin,sizeof(float *));
  if (m != NULL)
    for (l=0; l < nlin; l++)
      m[l]=(float *) calloc(ncol,sizeof(float));
  return(m);
}

void DestroiMatriz(float ***m, int nlin)
{
  int l;

  if (**m != NULL) {
    if (*m != NULL) {
      for (l=0; l < nlin; l++)
        free((*m)[l]);
      free(*m);
      *m = NULL;
    }
  }
}
/* ----------------------------------------------------------*/

void TranspoeMatriz(float m1[2][3], float m2[3][2]) /* passagem por
                                                       referência */
{
  int l,c;

  for (l=0; l < 3; l++)
    for (c=0; c < 2; c++)
```

```
      m2[l][c] = m1[c][l];
}

int main()
{
  float **m=NULL;
  float m1[2][3]={{1,2,3},{4,5,6}}; /* inicializa por linha */
  float m2[3][2];
  int l,c;

  m = CriaMatriz(2,3);
  printf("%d\n",m);
  DestroiMatriz(&m,2);
  printf("%d\n",m);
  /* ------------------------- */
  for (l=0; l < 2; l++) {
    for (c=0; c < 3; c++)
      printf("%f ",m1[l][c]);
    printf("\n");
  }
  TranspoeMatriz(m1,m2);
  for (l=0; l < 3; l++) {
    for (c=0; c < 2; c++)
      printf("%f ",m2[l][c]);
    printf("\n");
  }

  return 0;
}
```

Capítulo 14

Listas

14.1 Listas

Uma **lista** é uma seqüência dinâmica de elementos, denominados **nós**. Por exemplo, podemos usar uma lista para armazenar elementos de um conjunto, que podem ser inseridos e removidos do conjunto durante a execução do programa. Neste sentido, listas são dinâmicas; i.e., possuem tamanho variável durante a execução do programa. Portanto, os nós de uma lista são normalmente armazenados em regiões alocadas dinamicamente em memória e interligadas por apontadores. Para cada nó da lista, podemos ter um apontador para o nó seguinte, e a lista é dita **ligada simples**. Podemos também manter um apontador para o nó anterior, além do apontador para o próximo nó. Neste caso, a lista é dita **ligada dupla**. Neste curso veremos apenas os casos de listas ligadas simples.

A lista será representada por um apontador para o primeiro nó. Quando a lista estiver vazia, o apontador deve ser NULL. Podemos inserir novos nós no início, antes/após um dado elemento, ou no final da lista, **sempre cuidando para que o apontador da lista continue apontando para o primeiro nó**. A mesma observação vale para remoções no início, antes/após um dado elemento, e no final. O programa abaixo ilustra algumas operações com uma lista ligada simples.

```
#include <malloc.h>

typedef enum {false,true} bool;

typedef struct _no {
  int valor;
  struct _no *prox;
} No, Lista;

No    *CriaNo(int val); /* Cria e inicializa nó com valor em val */
void   DestroiLista(Lista **p); /* Destroi lista na memória */
bool   ListaVazia(Lista *p); /* Verifica se a lista está vazia */
void   InsereInicioLista(Lista **p, int val); /* Insere elemento no
                                                 início da lista */
void   ImprimeLista(Lista *p); /* Imprime elementos da lista */
bool   RemoveInicioLista(Lista **p, int *val); /* Remove elemento do
                                                   início da lista
```

```
                                                retornando em val o
                                                seu valor */

No  *CriaNo(int val)
{
  No *p;
  p        = (No *)calloc(1,sizeof(No)); /* a memória alocada em uma
                                            função, permanece alocada
                                            no programa até que seja
                                            liberada pelo comando free
                                            ou que o programa seja
                                            concluído. */
  p->valor = val;
  p->prox  = NULL;
  return(p);
}

bool ListaVazia(Lista *p)
{
  if (p == NULL)
    return(true);
  else
    return(false);
}

void DestroiLista(Lista **p)
{
  No *q;

  q = *p;
  while (!ListaVazia(q)){
    q = (*p)->prox;
    free(*p);
    *p = q;
  }
}

void InsereInicioLista(Lista **p, int val)
{
  No *q=NULL;

  q = CriaNo(val);
  if (ListaVazia(*p)){ /* lista vazia */
    *p = q;
  }else {
    q->prox = *p;
```

```
    *p      = q;
  }
}

bool RemoveInicioLista(Lista **p, int *val)
{
  Lista *q;

  if (!ListaVazia(*p)) { /* a lista não está vazia */
    *val = (*p)->valor;
    q    = (*p)->prox;
    free(*p);
    *p  = q;
    return(true);
  }else
    return(false);
}

void ImprimeLista(Lista *p)
{
  while (p != NULL){
    printf("%d ",p->valor);
    p = p->prox; /* estamos manipulando com cópia local de p, portanto
                    não estamos alterando seu conteúdo fora da
                    função. */
  }
  printf("\n");
}

int main()
{
  Lista *p=NULL; /* inicia vazia */
  int val;

  InsereInicioLista(&p,10);
  ImprimeLista(p);
  InsereInicioLista(&p,20);
  ImprimeLista(p);
  if (RemoveInicioLista(&p,&val))
      printf("%d foi removido\n",val);
  ImprimeLista(p);
  InsereInicioLista(&p,30);
  ImprimeLista(p);
  DestroiLista(&p);

  return 0;
```

```
}
```

Uma lista com inserção e remoção sempre em uma mesma extremidade, seja no início ou no final, é denominada **pilha** e é muito usada em vários algoritmos. Outra estrutura bastante útil é a **fila**. Neste caso, a inserção pode ser no início com remoção no final (ou a inserção pode ser no final e a remoção ser no início). Isto é, na fila, o primeiro elemento a ser inserido é o primeiro a ser removido (**FIFO**-*first-in-first-out*) e na pilha, o último a ser inserido é o primeiro a ser removido (**LIFO**-*last-in-first-out*).

14.2 Exercícios

Complete o programa acima criando funções para inserir e remover no final da lista; para consultar valores na lista; inserir e remover antes/após um dado elemento da lista; e para concatenar duas listas.

Capítulo 15

Arquivos

Até o momento, a entrada padrão (teclado) e a saída padrão (tela) foram utilizadas para a comunicação de dados. Muitas situações, porém, envolvem uma grande quantidade de dados, difícil de ser digitada e exibida na tela. Esses dados são armazenados em **arquivos** que ficam em dispositivos secundários de memória (e.g. disco rígido, CDROM, diskete). Portanto, um arquivo é uma seqüência de bytes com uma identificação que permite o sistema operacional exibir na tela seu nome, tamanho, e data de criação.

Existem dois tipos de arquivo: **arquivo texto** e **arquivo binário**. Um arquivo texto armazena os dados em ASCII, na forma de seqüência de caracteres, os quais podem ser exibidos na tela com o comando **more**, se o sistema operacional for unix/linux, ou com o comando **type**, se o sistema for dos. Um arquivo binário armazena os dados na forma binária (seqüência de bits), e normalmente ocupa bem menos espaço em memória do que um arquivo texto para armazenar a mesma informação.

Os arquivos são manipulados com variáveis do tipo apontador para arquivo, as quais são declaradas da seguinte forma:

```
int main()
{
  FILE *aparq;
}
```

Antes de ler ou escrever dados em um arquivo, precisamos endereçá-lo na variável aparq. Este endereçamento é realizado com o comando **fopen**, que pode abrir o arquivo para leitura e/ou escrita. Após leitura e/ou escrita, o comando **fclose** é usado para fechar o arquivo, liberando o apontador aparq.

15.1 Arquivo texto

A leitura de um arquivo texto é feita de forma similar aos casos de leitura da entrada padrão, usando agora comandos **fscanf**, **fgets**, e **fgetc** em vez de **scanf**, **gets**, e **getc**, respectivamente. A mesma observação vale para a gravação, agora usando os comandos **fprintf**, **fputs**, e **fputc**, em vez de **printf**, **puts**, e **putc**, respectivamente.

Por exemplo, suponha o arquivo **arq.txt** abaixo com número de amigos, nome e telefone de cada um, representando uma agenda pessoal.

```
11
Alexandre Falcao # 30
Paulo Miranda # 20
Felipe Bergo # 90
Fabio Capabianco # 25
Eduardo Picado # 15
Celso Suzuki # 19
Joao Paulo # 23
Jancarlo Gomes # 28
Danilo Lacerda # 18
Andre Goncalves # 35
Carlos Oliveira # 50
```

Um programa para ler este arquivo, ordenar os registros por nome, e gravar um arquivo ordenado é apresentado abaixo.

```
#include <stdio.h>

typedef struct _amigo {
  char nome[100];
  int  telefone;
} Amigo;

typedef struct _agenda{
  Amigo *amigo;
  int num_amigos;
} Agenda;

Agenda LeAgenda(char *nomearq);
void   OrdenaAgenda(Agenda a);
void   GravaAgenda(Agenda a, char *nomearq);
void   DestroiAgenda(Agenda a);

Agenda LeAgenda(char *nomearq)
{
  Agenda a;
  FILE *aparq;
  int i;
  char linha[200],*pos;

 aparq = fopen(nomearq,"r"); /* abre arquivo texto para leitura */
 fgets(linha,199,aparq); /* le cadeia de caracteres do arquivo atÃ© \n */
 sscanf(linha,"%d",&a.num_amigos);
 a.amigo = (Amigo *) calloc(a.num_amigos,sizeof(Amigo));
 if (a.amigo != NULL){
   for (i=0; i < a.num_amigos; i++) {
     fgets(linha,199,aparq);
```

```
     pos = strchr(linha,'#');
     strncpy(a.amigo[i].nome,linha,pos-linha-1);
     sscanf(pos+1,"%d",&a.amigo[i].telefone);
   }
 }

 fclose(aparq); /* fecha o arquivo */
 return(a);
}

void OrdenaAgenda(Agenda a)
{
  int i,j,jm;
  Amigo aux;

  for (j=0; j < a.num_amigos-1; j++){
    jm    = j;
    for (i=j+1; i < a.num_amigos; i++){
      if (strcmp(a.amigo[i].nome,a.amigo[jm].nome) < 0){
        jm    = i;
      }
    }
    if (j != jm){
      aux = a.amigo[j];
      a.amigo[j] = a.amigo[jm];
      a.amigo[jm] = aux;
    }
  }
}

void GravaAgenda(Agenda a, char *nomearq)
{
  int i;
  FILE *aparq;

  aparq = fopen(nomearq,"w"); /* cria novo arquivo texto aberto para escrita */
  fprintf(aparq,"%d\n",a.num_amigos); /* escreve no arquivo */
  for (i=0; i < a.num_amigos; i++)
    fprintf(aparq,"%s # %d\n",a.amigo[i].nome,a.amigo[i].telefone);
  fclose(aparq); /* fecha arquivo */
}

void DestroiAgenda(Agenda a)
{
  if (a.amigo != NULL)
    free(a.amigo);
```

```
}

int main()
{
  Agenda a;

  a = LeAgenda("arq.txt");
  OrdenaAgenda(a);
  GravaAgenda(a,"arq-ord.txt");
  DestroiAgenda(a);
  return(0);
}
```

15.2 Arquivo binário

No caso de arquivos binários, os comandos **fread** e **fwrite** são usados para leitura e escrita de seqüências de bytes. Se os arquivos de leitura e gravação do programa anterior fossem binários, as funções de leitura e gravação seriam conforme abaixo.

```
Agenda LeAgenda(char *nomearq)
{
  int i;
  FILE *aparq;
  Agenda a;

  aparq = fopen(nomearq,"rb"); /* abre arquivo binario para leitura */

  /* le 1 elemento do tamanho de um inteiro copiando sua sequencia de
     bytes para a memoria a partir do endereco apontado por
     &a.num_amigos. Avanca o respectivo numero de bytes na posicao para a
     proxima leitura. */

   fread(&a.num_amigos,sizeof(int),1,aparq);

   a.amigo = (Amigo *)calloc(a.num_amigos,sizeof(Amigo));
   for (i=0; i < a.num_amigos; i++){
     printf("%d\n",ftell(aparq)); /* So uma curiosidade. Diz a posicao no arquivo para leitura */
     fread(&a.amigo[i],sizeof(Amigo),1,aparq);
   }
   fclose(aparq); /* fecha arquivo */
return(a);
}

void GravaAgenda(Agenda a, char *nomearq)
{
  int i;
```

```
  FILE *aparq;

  aparq = fopen(nomearq,"wb"); /* cria novo arquivo binario aberto
  para escrita */

  /* escreve 1 elemento do tamanho de um inteiro na posicao atual de
     escrita no arquivo, copiando sua sequencia de bytes da memoria a
     partir do endereco apontado por &a.num_amigos. Avanca o respectivo
     numero de bytes na posicao para a proxima escrita. */

  fwrite(&a.num_amigos,sizeof(int),1,aparq);

  for (i=0; i < a.num_amigos; i++){
    fwrite(&a.amigo[i],sizeof(Amigo),1,aparq);
  }
  fclose(aparq); /* fecha arquivo */
}
```

Arquivos biários também podem ser modificados sem necessariamente terem que ser lidos e escritos por completo no disco. As funções abaixo ilustram uma alteração no número de telefone do segundo registro e a adição de um novo registro no final do arquivo.

```
void   AlteraRegistro(char *nomearq)
{
  FILE *aparq;
  Amigo aux;

/* abre arquivo existente para leitura/escrita */
  aparq = fopen(nomearq,"r+");

/* posiciona leitura/escrita no inicio do segundo registro */
  fseek(aparq,sizeof(int)+sizeof(Amigo),SEEK_SET);

  fread(&aux,sizeof(Amigo),1,aparq); /* le nome e telefone */
  aux.telefone = 10; /* Altera telefone para o novo numero */

/* posiciona leitura/escrita no inicio do segundo registro */
  fseek(aparq,sizeof(int)+sizeof(Amigo),SEEK_SET);

  fwrite(&aux,sizeof(Amigo),1,aparq); /* grava registro modificado */
  fclose(aparq); /* fecha arquivo */
}

void   AdicionaRegistro(char *nomearq)
{
  FILE *aparq;
```

```
  Amigo aux;
  int num_amigos;

/* abre arquivo existente para leitura/escrita. Posiciona escrita no
   final do arquivo, mas permite leitura dos registros se o apontador for
   reposicionado. So permite escrita no final.*/
  aparq = fopen(nomearq,"a+");

  sprintf(aux.nome,"Gloria Meneses");
  aux.telefone = 65;
  fwrite(&aux,sizeof(Amigo),1,aparq); /* Grava no final */
  fseek(aparq,0,SEEK_SET); /* posiciona leitura no inicio do arquivo */
  fread(&num_amigos,sizeof(int),1,aparq); /* le numero de amigos */
  num_amigos ++; /* atualiza o numero de amigos */
  fclose(aparq); /* fecha arquivo */
/* abre para leitura/escrita em qualquer posicao */
  aparq = fopen(nomearq,"r+");

/* grava numero de amigos atualizado */
  fwrite(&num_amigos,sizeof(int),1,aparq);
  fclose(aparq); /* fecha arquivo */
}
```

As funções acima gravam o arquivo binário com registros de tamanho fixo. Para o exemplo dado, o arquivo binário ocupa mais espaço do que o arquivo texto. Isto porque estamos gravando 50 bytes do nome de cada amigo, quando este nome ocupa menos que 50 bytes. Uma alternativa é gravar 1 byte com a informação do comprimento do nome e depois o nome. Isto reduz bastante o tamanho do arquivo binário.

```
Agenda LeAgenda(char *nomearq)
{
  Agenda a;
  FILE *aparq;
  int i;
  unsigned char comp;

 aparq = fopen(nomearq,"rb");
 fread(&a.num_amigos,sizeof(int),1,aparq);
 a.amigo = (Amigo *) calloc(a.num_amigos,sizeof(Amigo));
 if (a.amigo != NULL){
   for (i=0; i < a.num_amigos; i++) {
     fread(&comp,sizeof(unsigned char),1,aparq);
     fread(a.amigo[i].nome,comp,1,aparq);
     fread(&a.amigo[i].telefone,sizeof(int),1,aparq);
   }
 }
```

```
 fclose(aparq);
 return(a);
}

void GravaAgenda(Agenda a, char *nomearq)
{
  int i;
  unsigned char comp;
  FILE *aparq;

  aparq = fopen(nomearq,"wb");
  fwrite(&a.num_amigos,sizeof(int),1,aparq);
  for (i=0; i < a.num_amigos; i++){
    comp = (unsigned char)strlen(a.amigo[i].nome);
    fwrite(&comp,sizeof(unsigned char),1,aparq);
    fwrite(a.amigo[i].nome,comp,1,aparq);
    fwrite(&a.amigo[i].telefone,sizeof(int),1,aparq);
  }
  fclose(aparq);
}
```

Capítulo 16

Sistema e programas

16.1 Sistema

Um sistema é um conjunto de programas que compartilham uma ou mais bibliotecas de funções. Quando projetamos um sistema, cada estrutura abstrata e as funções que realizam operações com esta estrutura devem ser declaradas em arquivos com extensões .h e .c separados dos arquivos de programa. Por exemplo, nos arquivos contorno.h e matriz.h, nós declaramos estruturas e protótipos das funções.

1. Arquivo contorno.h

```
#ifndef CONTORNO_H
#define CONTORNO_H 1

#include <malloc.h>

typedef struct _ponto
{
  float x,y;
} Ponto;

typedef struct _contorno
{
  Ponto *p; /* vetor de pontos */
  int n;    /* numero de pontos */
} Contorno;

Contorno CriaContorno(int n); /* Aloca espaco para contorno com n pontos */
void     DestroiContorno(Contorno c); /* Desaloca espaco do contorno */

/* Aqui voce pode acrescentar os prototipos de outras
   funcoes que manipulam Contorno */

#endif
```

2. Arquivo matriz.h

```
#ifndef MATRIZ_H
#define MATRIZ_H 1

#include <malloc.h>

typedef struct _matriz {
  float **elem; /* elementos da matriz */
  int nlin,ncol; /* numero de linhas e colunas */
} Matriz;

Matriz CriaMatriz(int nlin, int ncol); /* Aloca espaco para matriz nlin x ncol */
void   DestroiMatriz(Matriz m); /* Desaloca espaco */

/* Aqui voce pode acrescentar os prototipos de outras
   funcoes que manipulam Matriz */


#endif
```

Nos arquivos contorno.c e matriz.c, nós incluímos os respectivos arquivos .h e declaramos os escopos dessas funções.

1. Arquivo contorno.c

```
#include "contorno.h"

Contorno CriaContorno(int n)
{
  Contorno c;

  c.n = n;
  c.p = (Ponto *)calloc(n,sizeof(Ponto));
  return(c);
}

void     DestroiContorno(Contorno c)
{
  if (c.p != NULL) {
    free(c.p);
    c.p = NULL;
  }
}

/* Aqui voce acrescenta os escopos das outras funcoes
   que manipulam Contorno */
```

2. Arquivo matriz.c

```
#include "matriz.h"

Matriz CriaMatriz(int nlin, int ncol)
{
  Matriz m;
  int i;

  m.nlin = nlin;
  m.ncol = ncol;
  m.elem = (float **)calloc(nlin,sizeof(float *));
  if (m.elem != NULL)
  for (i=0; i < nlin; i++)
    m.elem[i] = (float *)calloc(ncol,sizeof(float));
  return(m);
}

void DestroiMatriz(Matriz m)
{
  int i;

  if (m.elem != NULL){
    for (i=0; i < m.nlin; i++)
      free(m.elem[i]);
    free(m.elem);
    m.elem = NULL;
  }
}

/* Aqui voce acrescenta os escopos das outras funcoes
   que manipulam Matriz */
```

Para organizar o sistema, nós podemos criar uma estrutura de diretórios com raiz mc102, por exemplo, colocando os arquivos com extensão .h em mc102/include e os arquivos com extensão .c em mc102/src. Como um arquivo com extensão .c não possue a função main, ele não é considerado programa. Seu código após compilação tem extensão .o e é denominado **código objeto** (não executável). Os respectivos arquivos com extensão .o podem ser colocados em mc102/obj. Para que as estruturas e funções desses arquivos sejam disponibilizadas para uso por programas, os arquivos objeto devem ser "ligados" em um arquivo com extensão .a, i.e. arquivo biblioteca de funções. O *script* **Makefile** abaixo realiza esta tarefa colocando a biblioteca libmc102.a no diretório mc102/lib. O Makefile fica no diretório mc102.

```
LIB=./lib
INCLUDE=./include
BIN=./
```

```
SRC=./src
OBJ=./obj
#FLAGS= -g -Wall
FLAGS= -O3 -Wall

libmc102: $(LIB)/libmc102.a
echo "libmc102.a construida com sucesso..."

$(LIB)/libmc102.a: \
$(OBJ)/contorno.o \
$(OBJ)/matriz.o

ar csr $(LIB)/libmc102.a \
$(OBJ)/contorno.o \
$(OBJ)/matriz.o

$(OBJ)/contorno.o: $(SRC)/contorno.c
gcc $(FLAGS) -c $(SRC)/contorno.c -I$(INCLUDE) \
-o $(OBJ)/contorno.o

$(OBJ)/matriz.o: $(SRC)/matriz.c
gcc $(FLAGS) -c $(SRC)/matriz.c -I$(INCLUDE) \
-o $(OBJ)/matriz.o

apaga:
rm $(OBJ)/*.o;

apagatudo:
rm $(OBJ)/*.o; rm $(LIB)/*.a
```

16.2 Programas

Todo programa que quiser usar as estruturas e funções da biblioteca libmc102.a devem incluir os arquivos .h da biblioteca e devem ser compilados com o comando abaixo. Suponha, por exemplo, que o programa prog.c abaixo está no mesmo nível do diretório mc102.

1. Programa prog.c

```
#include "contorno.h"
#include "matriz.h"

int main()
{
  Contorno c;
  Matriz m;
```

```
    c = CriaContorno(100);
    m = CriaMatriz(5,9);

    DestroiContorno(c);
    DestroiMatriz(m);
    return(0);
  }
```

2. Comando de compilação

```
gcc prog.c -I./mc102/include -L./mc102/lib -o prog -lmc102
```

16.2.1 Argumentos de programa

Em uma das aulas anteriores assumimos um nome fixo para os arquivos lidos e gravados em disco.

```
int main()
{
  Agenda a;

  a = LeAgenda("arq.txt");
  OrdenaAgenda(a);
  GravaAgenda(a,"arq-ord.txt");
  DestroiAgenda(a);
  return(0);
}
```

Porém, seria mais conveniente se esses nomes fossem passados como argumentos do programa na linha de execução (e.g. **programa arq.txt arq-ord.txt**). Isto é possível, pois a função main pode ser declarada com dois argumentos: int main(int argc, char **argv). O valor de **argc** é o número de argumentos em argv e **argv** é um vetor de *strings*, onde cada *string* contém um argumento do programa (e.g. pode ser um número real, inteiro, um caracter, ou uma cadeia de caracteres). No caso do programa acima, teríamos:

```
int main(int argc, char **argv)
{
  Agenda a;

  if (argc != 3) {
    printf("uso: %s <entrada> <saida>\n",argv[0]);
    exit(-1);
  }
  a = LeAgenda(argv[1]);
  OrdenaAgenda(a);
  GravaAgenda(a,argv[2]);
  DestroiAgenda(a);
  return(0);
}
```

Nos casos de argumentos inteiros e reais, as funções **atoi** e **atof** podem ser usadas para converter *string* nesses valores, respectivamente.

```
#include "contorno.h"
#include "matriz.h"

int main(int argc, char **argv)
{
  Contorno c;
  Matriz m;

  if (argc != 4){
    printf("uso: %s <numero de pontos> <numero de linhas> <numero de colunas>\n",argv[0]);
    exit(-1);
  }

  c = CriaContorno(atoi(argv[1]);
  m = CriaMatriz(atoi(argv[2]),atoi(argv[3]));

  DestroiContorno(c);
  DestroiMatriz(m);
  return(0);
}
```