Linguagem de Programação C

Notas de Aulas

Parte 3:
Estruturas e Arquivos

Professora: Luciana Rita Guedes

Versão atual: julho/2022

Índice

1. ESTRUTURAS EM C

OBJETIVO:

- **Usar esquemas especiais de dados para simplificar tarefas de programação. Estes esquemas são chamados de *estruturas*. As *estruturas* constituem-se num método de armazenagem desenhado pelo próprio programador.**

1.1 ESTRUTURAS SIMPLES:

- **Uma *estrutura* é uma coleção de uma ou mais variáveis agrupadas sob um único nome;**
- **As variáveis de uma *estrutura*, ao contrário de uma matriz, podem ser de diferentes tipos de dados;**
- **Uma *estrutura* pode conter quaisquer tipos de dados, inclusive matrizes ou mesmo outras *estruturas*.**

Definindo e Declarando Estruturas Simples:

- **Para definir uma *estrutura simples* usa-se a seguinte sintaxe:**

```
struct rotulo
  {
    membros_da_estrutura;
  };
```

  - **A instrução acima define uma *estrutura* chamada *rotulo* que pode conter diversos membros, os quais podem ser do mesmo tipo ou podem ser de tipos diferentes;**

Exemplos de Definição de Estruturas:

```
struct hora
  {
    int horas;
    int minutos;
    int segundos;
   };
```

```
struct data
  {
    char dia[2];
    char mes[2];
    char ano[4];
 };
```

- **Para declarar *estruturas* cujo "gabarito" foi previamente definido usa-se seguinte sintaxe:**

```
struct rotulo instancia1, instancia2,...;
```

  - **A instrução acima declara *instancias* da estrutura chamada *rotulo*, isto é, variáveis cujo "formato" é definido pela estrutura denominada *rotulo*;**

- **Outra forma de declarar *estruturas* é no momento de criação da estrutura (definição) colocando-se os nomes das instâncias logo após à definição:**

```
struct rotulo
  {
    membros_da_estrutura;
  }  instancia1, instancia2,...;
```

Exemplos de Declaração de Estruturas:

```
// declaração em separado
struct hora
  {
    int horas;
    int minutos;
   };
// pode haver outras
// instrucoes aqui
struct hora inicio, fim;
```

```
// declaração junto c/definição
struct data
  {
    char dia[2];
    char mes[2];
    char ano[4];
   } data_nasc, data_atual;
// neste caso a declaracao deve
// vir logo apos a definicao
```

1.1.1 Acessando os Membros de Uma Estrutura:

- **Os membros individuais de uma *estrutura* podem ser acessados como qualquer variável, desde que seja usado o *operador de membro de estrutura* (.), entre o nome da estrutura (instância) e o nome do membro. O *operador de membro de estrutura* também é chamado de *operador ponto*:**

```
nome_instancia.nome_membro = valor;
printf( ... ,nome_instancia.nome_membro);
```

Exemplos de Uso de Membros de Estruturas:

```
// definição da estrutura HORA
struct hora
  {
    int horas;
    int minutos;
   };
// pode haver outras instrucoes aqui
// declaração das instâncias INICIO e FIM
struct hora inicio, fim;
// uso dos membros da estrutura através da instancia INICIO
inicio.horas = 10;
inicio.minutos = 35;
// uso dos membros da estrutura através da instancia FIM
fim.horas = inicio.horas;
fim.minutos = 49;
```

- **Ainda com relação ao uso dos membros de uma estrutura, é possível copiar informações entre estruturas de mesmo formato, como no exemplo abaixo:**

```
// os dados contidos nos membros da estrutura INICIO são
// passados a cada um dos membros da estrutura FIM
fim = inicio;
```

1.2 ESTRUTURAS QUE CONTÊM ESTRUTURAS:

- **Uma *estrutura* pode conter outras estruturas dentro de si formando um agrupamento de variáveis mais complexos e interessantes. Vejamos isto através de um exemplo:**

Exemplos de Estruturas que Contêm Estruturas:

```
// definição da estrutura COORD que é composta de
// duas coordenadas num plano cartesiano (X e Y)
struct coord
  {
    int x;
    int y;
   };
// definição e declaração da estrutura RETANGULO composta
// de dois conjuntos de coordenadas X e Y
struct retangulo
  {
    struct coord superior_esquerdo;
    struct coord inferior_direito;
   }  caixa;
// uso dos membros da estrutura da instancia CAIXA
caixa.superior_esquerdo.x = 0;
caixa.superior_esquerdo.y = 10;
caixa.inferior_direito.x = 100;
caixa.inferior_direito.y = 200;
// estes dados podem ser interpretados como um conjunto
// de duas coordenadas X e Y no plano cartesiano para
// formarem uma caixa retangular
```

- **O resultado da atribuição acima pode ser representado da seguinte forma:**

CAIXA			
SUPERIOR_ESQUERDO		INFERIOR_DIREITO	
X	Y	X	Y
0	10	100	200

Programa Exemplo com Estruturas que Contêm Estruturas:

```
01: /* Demonstra o uso de estruturas que contem outras estruturas */
02:
03: /* Recebe informacoes sobre as coordenadas dos cantos de
04:    um retangulo e calcula a sua area. Presume que a
05:    coordenada y do canto superior esquerdo e maior que a
06:    coordenada y do canto inferior direito, que a coordenada x
07:    do canto inferior direito e maior que a coordenada x do canto
08:    superior esquerdo, e que todas as coordenadas sao positivas.
      */
09:
10: #include <stdio.h>
11:
12: int comprimento, largura;
13: long area;
14:
15: struct coord {
16:      int x;
17:      int y;
18:    };
19:
20: struct retangulo {
21:     struct coord sup_esq;
22:     struct coord inf_dir;
23:   } caixa;
24:
25: main()
26: {
27:      /* Recebe as coordenadas */
28:
29:      printf("\nDigite a coordenada x superior esquerda: ");
30:      scanf("%d", &caixa.sup_esq.x);
31:
32:      printf("\nDigite a coordenada y superior esquerda: ");
33:      scanf("%d", &caixa.sup_esq.y);
34:
35:      printf("\nDigite a coordenada x inferior direita: ");
36:      scanf("%d", &caixa.inf_dir.x);
37:
38:      printf("\nDigite a coordenada y inferior direita: ");
39:      scanf("%d", &caixa.inf_dir.y);
40:
41:      /* Calcula o comprimento e a largura */
42:      largura = caixa.inf_dir.x - caixa.sup_esq.x;
43:      comprimento = caixa.inf_dir.y - caixa.sup_esq.y;
44:
45:      /* Calcula e informa a area */
46:      area = largura * comprimento;
47:      printf("\nA área do retângulo é de %ld unidades.", area);
48: }
```

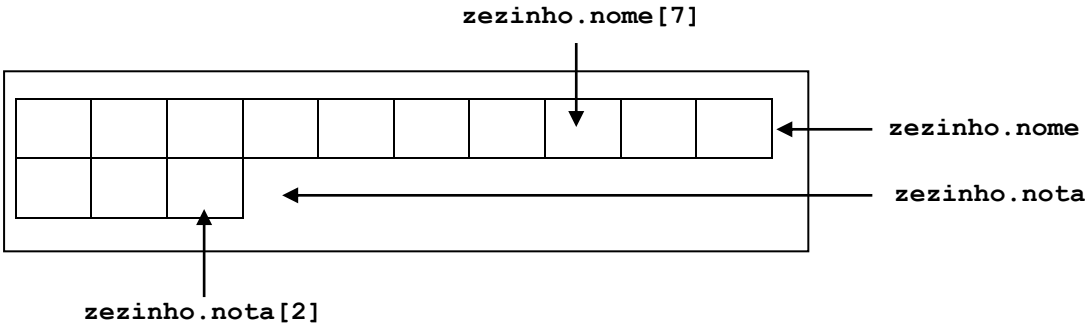
1.3 ESTRUTURAS QUE CONTÊM MATRIZES:

- **Pode-se definir uma *estrutura* cujos membros incluam uma ou mais matrizes. A matriz pode ser de qualquer tipo válido em C. Vejamos isto através de um exemplo:**

Exemplo de Estruturas que Contêm Matrizes:

```
// definição da estrutura ALUNO contendo duas matrizes
struct aluno
  {
    char nome[10];
    float nota[3];
   };
// declaração de uma instância da estrutura ALUNO
struct aluno ZEZINHO;
```

- **A estrutura definida acima poderia ser representada da seguinte forma:**



- **Para acessar os membros da estrutura, usa-se as mesmas instruções de manipulação de matrizes:**

```
zezinho.nota[2] = 10;
zezinho.nome[1] = ´J´;
```

Exemplo de programa com estrutura que contêm matrizes:.

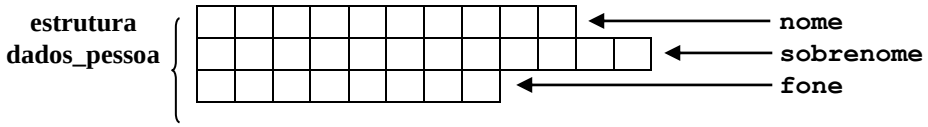
```
01: /* Demonstra uma estrutura que contem matrizes como membros */
02:
03: #include <stdio.h>
04:
05: /* Define e declara uma estrutura para conter os dados, */
06: /* cujos membros sao uma variavel float e duas matrizes char. */
07:
08: struct dados {
09:    float quantia;
10:    char nome[30];
11:    char snome[30];
12:  } reg;
13:
14: main()
15: {
16:     /* Recebe os dados do teclado. */
17:
18:     printf("Digite o nome e o sobrenome do doador,\n");
19:     printf("separados por um espaco: ");
20:     scanf("%s %s", reg.nome, reg.snome);
21:
22:     printf("\nDigite a quantia doada: ");
23:     scanf("%f", &reg.quantia);
24:
25:     /* Exibe as informacoes. */
26:     /* Nota: %.2f especifica um valor de ponto flutuante */
27:     /* que deve ser impresso com dois algarismos a direita */
28:     /* do ponto decimal. */
29:
30:     /*Exibe os dados na tela */
31:
32:     printf("\nO doador %s %s deu $%.2f.", reg.nome, reg.snome,
                reg.quantia);
33:
34: }
```

1.4 MATRIZES DE ESTRUTURAS:

- **As matrizes de estruturas são ferramentas de programação poderosas. Significa que, partindo de uma estrutura definida, pode-se ter uma matriz onde cada elemento é uma estrutura deste tipo.**
- **Por exemplo, supondo uma estrutura como declarada abaixo:**

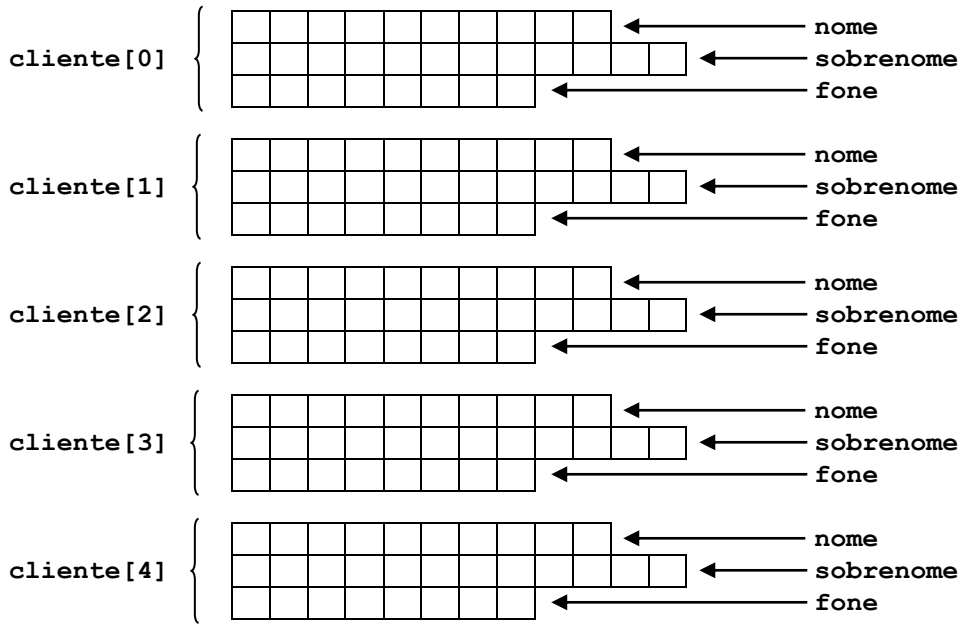
```
struct dados_pessoa
    {
      char nome[10];
      char sobrenome[12];
      char fone[8];
    };
```

- **Esta estrutura poderia ser representada da seguinte forma:**



- **Para definir uma matriz a partir desta estrutura poderia ser usada, por exemplo, a seguinte definição:**

```
struct dados_pessoa cliente[5];
```



Exemplo de programa com matrizes de estruturas:

```
01: * Demonstra o uso de matrizes de estruturas */
02:
03: #include <stdio.h>
04: #define tam 5
05:
06: main()
07: {
08:    struct dados_aluno
09:     {
10:       char  nome[15];
11:       float media;
12:       int   ano_nasc;
13:     } aluno[tam];
14:
15:     int x;
16:
17:     /* Recebe os dados do teclado. */
18:
19:     for(x=0;x<tam;x++)
20:     {
21:        printf("\nDigite o nome do %io. aluno: ",x+1);
22:        scanf("%s", aluno[x].nome);
23:        printf("Digite a média deste aluno: ");
24:        scanf("%f", &aluno[x].media);
25:        printf("Digite o ano de nascimento deste aluno: ");
26:        scanf("%i", &aluno[x].ano_nasc);
27:     }
28:
29:     /*Exibe os dados na tela */
30:
31:     printf("\n\tNomes          \tMedias\tIdades");
32:
33:     for(x=0;x<tam;x++)
34:     {
35:       printf("\n\t%s\t%.2f\t%i",aluno[x].nome,aluno[x].media,
                2001-aluno[x].ano_nasc);
36:     }
37:
38: }
```

1.5 INICIALIZANDO ESTRUTURAS:

- **As estruturas podem ser inicializadas ao serem declaradas. O procedimento é semelhante ao adotado para inicializar matrizes: coloca-se um sinal de igualdade após a declaração da estrutura seguido de uma lista de valores separados por vírgula e delimitados por chaves. Por exemplo:**

```
struct venda
  {
    char  cliente[10];
    char  item[10];
    float quantidade;
  } ultima_venda = {"Fulano","Rebimboca",10.5};
```

- **No caso de uma estrutura que contenha outras estruturas como membros, os valores de inicilização devem ser listados em ordem. Esses valores serão armazenados nos membros da estrutura no ordem em que os membros estão listados na definição da estrutura. Por exemplo:**

```
struct cliente
  {
    char  empresa[20];
    char  contato[15];
};

struct venda
  {
    struct cliente comprador;
    char  item[10];
    float quantidade;
  } primeira_venda = { {"Indústria
Acme","Fulano"},
                       "Rebimboca",
                       10.5
                     };
```

- **No caso de uma *matriz de estrutura*, os dados de inicilização fornecidos serão aplicados, pela ordem, às estruturas da matriz. Por exemplo:**

```
struct cliente
  {
    char  empresa[20];
    char  contato[15];
};

struct venda
  {
    struct cliente comprador;
    char  item[10];
    float quantidade;
  } diaria[10] = {
                   {  /* Dados para diaria[0] */
                      {"Indústria Acme","Fulano"},
                      "Rebimboca",
                      10.5
                   }
                   {  /* Dados para diaria[1] */
                      {"ABC Ltda.","Beltrano"},
                      "Parafuseta",
                      5.6
                   }
                   {  /* Dados para diaria[2] */
                      {"Cia. XYZ","Cicrano"},
                      "Treco",
                      17.0
                   }
                  };
```

- **No exemplo acima, apenas os três primeiros elementos da matriz de estrutura *diaria* receberam informações. Os demais elementos não foram inicializados.**

2. ESTRUTURAS E PONTEIROS

- **Pode-se usar ponteiros como membros de estruturas e também declarar ponteiros que apontem para estruturas.**

2.1 Ponteiros Como Membros de Estruturas:

- **Para usar um ponteiro como membro de uma estrutura, para declará-los precedidos do operador de indireção (*), tal como na declaração comum de ponteiros. Por exemplo:**

```
struct financiamento
  {
      int   *valor;
      float *juros;
  }fin_automovel;
```

- **Estas instruções declaram uma estrutura denominada *financiamento* que possui dois membros que são ponteiros. Além disso, uma estrutura denominada *fin_automovel* é definida com este "gabarito" de *financiamento*.**
- **A inicialização dos ponteiros acima declarados pode ser, por exemplo:**

```
fin_automovel.valor = &custo_total;
fin_automovel.juros = &taxa;
```

- **Também pode-se usar o operador de indireção para acessar o valor das variáveis apontadas por estes ponteiros:**

```
printf("\nValor: %d", *fin_automovel.valor);
tot = *fin_automovel.valor +
*fin_automovel.juros;
```

2.2 Ponteiros para Strings como Membros de Estruturas:

- **O uso mais comum de ponteiros como membros de estruturas é o de ponteiros para strings, como no exemplo abaixo:**

```
struct mensagem
  {
      char *msg1;
      char *msg2;
  } observacao;
```

  - **Tal como no uso comum de ponteiros para strings, pode-se inicializá-lo diretamento com um string, como abaixo:**

```
observação.msg1="Primeira mensagem de
observação";
observação.msg2="Segunda mensagem de observação";
```

  - **Desta mesma forma, os ponteiros que são membros de estruturas podem ser usados em qualquer lugar em que um ponteiro normal poderia ser usado. Por exemplo:**

```
printf("%s %s",observação.msg1, observação.msg2)

puts(observação.msg1);
puts(observação.msg2);
```

2.3 Ponteiros para strings x Matrizes de *char* em Estruturas:

- **Tanto o uso de ponteiros para strings quanto o uso de matrizes para o tipo *char* são usados para "armazenar" um string dentro de uma estrutura.**

Definição de Ponteiros para Strings x Matrizes do tipo char:

```
struct msg
  {
    char p1[20];
    char *p2;
   } obs;
```

Uso de Ponteiros para Strings x Matrizes do tipo char:

```
/* atribuição de valores */
strcpy(obs.p1,"Primeira mensagem");
strcpy(obs.p2,"Segunda mensagem");
/* impressão dos valores */
puts(obs.p1);
puts(obs.p2);
```

- **Observe que a forma de uso tanto de ponteiros para *strings* quanto de matrizes para o tipo *char* é idêntico.**

- **Assim, a diferença entre ambos consiste basicamente em:**

  - **Ao definir uma estrutura que contenha uma matriz do tipo *char*, todas as instâncias desse tipo de estrutura conterão espaço de armazenagem para o tamanho especificado.**
  - **Além disso, com o uso de matrizes para o tipo *char*, o tamanho do *string* estará limitado ao tamanho**

especificado e não poderá armazenar um número de caracteres maior na estrutura.

Exemplo: limitação no uso de matrizes para o tipo char:

```
// definição e declaração
struct nome_completo
{
   char prenome[10];
   char sobrenome[10];
} cliente;
// pode haver outras instruções aqui

strcpy(cliente.prenome,"Escolástica");
/* string maior que matriz – provocará erros */

strcpy(cliente.sobrenome,"Silva");
/* string menor que matriz – desperdício de memória */
```

- **O uso de ponteiros para *strings* impede que tais problemas venham a ocorrrer.**

- **Vantagens do uso de ponteiros para *strings* no lugar de matrizes do tipo *char*:**

  - **Cada instância da estrutura conterá apenas o espaço de armazenagem necessário para o próprio ponteiro;**
  - **Os *strings* propriamente ditos serão armazenados em qualquer outra parte da memória (não há necessidade de se saber onde);**
  - **Não há restrição ao tamanho do *string* nem haverá espaço ocioso;**
  - **Cada ponteiro da estrutura pode apontar para um string de qualquer tamanho;**
  - **Os *strings* referenciados por estes ponteiros passam a fazer parte da estrutura indiretamente.**

2.4 PONTEIROS PARA ESTRUTURAS:

- **É possível declarar e usar ponteiros que apontem para estruturas exatamente como são usados para qualquer outro tipo de armazenagem de dados.**
- **As grandes utilidades deste tipo de ponteiro são:**
  - **para passar uma estrutura para uma função;**
  - **para usá-los em *listas encadeadas*.**

Exemplo do uso de Ponteiros para Estruturas:

```
/* definição e declaração da estrutura */
struct produto
  {
    int  codigo;
    char nome[10];
   } prod1;

/* declaração de um ponteiro para esta estrutura
*/
struct produto *p_prod;

/* inicialização do ponteiro */
p_prod = &prod1;

/* atribuição de valor através da indireção */
(*p_prod).codigo = 100;  // prod1.codigo = 100;

strcpy((*p_prod).nome,"Rebimboca");

/* operador (.) tem prioridade sobre (*) */

/* impressão dos valores através da indireção */
printf("%d\n",(*p_prod).codigo);
printf("%s\n",(*p_prod).nome);
puts((*p_prod).nome);
```

- **Outra forma de acessar membros da estrutura através de ponteiros é com o uso do *operador de acesso a membro* (->) como mostrado a seguir:**

```
printf("%d\n",p_prod->codigo);
puts(p_prod->nome);
```

`p_prod->codigo` é similar a `(*p_prod).codigo`

```
struct produto
{
    int  codigo;
    char nome[10];
} prod[100];

/* declaração de um ponteiro para esta estrutura */
struct produto *p_prod;

p_prod = &prod[0];  //  p_prod = prod;
// usando a matriz prod
for(i=0;i<100;i++)
{
    printf("Digite o código do %iº produto:");
    scanf("%d",&prod[i].codigo)
    printf("Digite o nome do %iº produto:");
    scanf("%s",prod[i].nome)
}

// usando o ponteiro p_prod
for(i=0;i<100;i++)
{
    printf("Digite o código do %iº produto:");
    scanf("%d",&p_prod->codigo)
    printf("Digite o nome do %iº produto:");
    scanf("%s",p_prod->nome)
    p_prod++;
}  // no final, o ponteiro estará apontando para
   // o fim da matriz (um endereço após o último)
```

PONTEIROS E MATRIZES DE ESTRUTURAS:

```
01: /* Demonstra a progressao atraves de unia matriz de estruturas
*/
02: /* usando a notacao de ponteiros. */
03:
04: #include <stdio.h>
05:
06: #define MAX 4
07:
08: /* Define uma estrutura, depois declara e inicializa */
09: /* uma matriz contendo 4 estruturas. */
10:
11: struct peca {
12:    int numero;
13:    char nome[10];
14:  } dados[MAX] = {1, "Smith",
15:                  2, "Jones",
16:                  3, "Adams",
17:                  4, "Wilson"
18:                 };
19:
20: /* Declara um ponteiro p/o tipo peca e uma variavel de contagem
*/
21:
22: struct peca *p_peca;
23: int contagem;
24:
25: main()
26: {
27:   /* Inicializa o ponteiro para o primeiro elemento da matriz */
28:
29:   p_peca = dados;
30:
31:   /* Faz um loop atraves da matriz, incrementando o ponteiro */
32:   /* a cada iteracao. */
33:
34:   for (contagem = 0; contagem < MAX; contagem++)
35:   {
36:      printf("\nNo endereco %d: %d %s",p_peca, p_peca—>numero,
                                          p_peca—>nome);
37:      p_peca++;
38:
39:   }
40:
41: }
```

PONTEIROS E MATRIZES DE ESTRUTURAS:

```
01: /* Demonstra o uso de estrutura como argumento de uma função */
02:
03: #include <stdio.h>
04:
05: /* Define e declara uma estrutura para conter os dados */
06:
07: struct dados {
08:    float quantia;
09:    char  nome[30];
10:    char  snome[30];
11:  } reg;
12:
13: /* O protótipo da função: A função não tem valor de retorno e */
14: /* aceita como único argumento uma estrutura do tipo 'dados'. */
15:
16: void print_reg(struct dados x);
17:
18: main()
19: {
20:    /* Recebe os dados do teclado. */
21:
22:    printf("Digite o nome e o sobrenome do doador,\n");
23:    printf("separados por um espaço: ");
24:    scanf("%s %s", reg.nome, reg.snome);
25:
26:    printf("\nDigite a quantia doada: ");
27:    scanf("%f", &reg.quantia);
28:
29:    /* Chama a função que exibe os dados. */
30:
31:    print_reg( reg );
32: }
33:
34: void print_reg(struct dados x)
35: {
36:    printf("\nO doador %s %s deu $%.2f.", x.nome, x.snome,
                                             x.quantia);
37: }
```

```
Digite o nome e o sobrenome do doador, separados por um espaço:
Carlos Silva Digite a quantia doada: 1000.00
O doador Carlos Silva deu $1000.00.
```

3. ARQUIVOS EM DISCO

ARQUIVO (conceito):
"Uma coleção de bytes armazenados em memória secundária (disquete, disco rígido, fita magnética, etc.), e referenciados por um nome comum."

IBPI, Dominando a linguagem C

- **Os dados manipulados em C, a princípio, vêm do teclado (dispositivo padrão de entrada STDIN) e são enviados para a tela (dispositivo padrão de saída STDOUT).**

- **A linguagem C permite que sejam manipulados outros dispositivos de entrada e saída, como a impressora (STDPRN) ou mesmo arquivos em disco.**

- **Os arquivos em disco manipulados pela linguagem C podem ser do tipo *texto* ou do tipo *binário*.**

ARQUIVO EM MODO DE TEXTO:

- **Um *arquivo em modo de texto* é uma seqüência de linhas, onde cada linha contém zero ou mais caracteres e termina com caracteres de sinalização de "fim de linha".**
- **Uma linha não é um string pois não termina com "\0" e sim com "\n".**
- **Um "\n" é um sinalizador de "fim de linha" e, em geral, é representado pelos caracteres CR (carriage return) e LF (line feed).**

ARQUIVO EM MODO BINÁRIO:

- **Um *arquivo em modo binário* é uma seqüência de caracteres onde os dados são gravados e lidos exatamente como estão armazenados na memória** (2 bytes para inteiros, 4 bytes para float e assim por diante).

NOMES DE ARQUIVOS:

- **Ao lidar com arquivos é preciso mencionar o nome do arquivo desejado.**

- **Os nomes de arquivos são armazenados em strings e são idênticos aos nomes usados pelo sistema operacional, seguindo portanto as mesmas regras de nomenclatura.**

- **Nomes de arquivos podem conter também informações sobre o drive e o diretório onde se encontra.**

- **Como exemplo, no DOS, poderíamos ter o arquivo:**

```
C:\DATA\LIST.TXT
```

- **Num programa em C, este mesmo arquivo poderia ser mencionado da seguinte forma:**

```
char *nomearq = "C:\\DATA\\LIST.TXT";
```

Observações:

- **o uso de duas barras invertidas consecutivas deve-se ao fato que, em C, uma barra invertida possui um significado especial quando colocada dentro de um string;**

- **porém, se o nome do arquivo for especificado via teclado, deve-se digitar somente uma barra invertida.**

- **no DOS, a regra para nomenclatura de arquivos permite um nome e uma extensão de até três caracteres; os caracteres válidos são:**

```
- letras de A a Z
- números de 0 a 9
- caracteres especiais como _ ! @ $ entre
outros.
```

3.1 A ESTRUTURA FILE:

- **O tipo FILE é uma estrutura declarada em** <stdio.h> **cujos elementos são informações sobre o arquivo que está sendo acessado, tais como: status do arquivo, endereço do buffer para transferência de dados, posição corrente do ponteiro, entre outras.**
- **Cada arquivo manipulado pela linguagem C tem a ele associada uma estrutura do tipo FILE e o modo de referência ao arquivo passa a ser através de um ponteiro para essa estrutura:**

```
FILE *fptr;  /* ponteiro para um arquivo */
```

- ***fptr* é um ponteiro para a estrutura FILE que será usado para referenciar o arquivo desejado dentro do programa.**

3.2 ABRINDO ARQUIVOS:

- **Para manipular arquivos na linguagem C é necessário, primeiramente, *abrir o arquivo*, ou seja, deixá-lo disponível para leitura e/ou gravação.**
- **Para abrir um arquivo em C utiliza-se a função *fopen*, que devolve um ponteiro para a estrutura FILE associada ao arquivo:**

```
fptr = fopen(nomearq,modo);
```

onde
fptr: é o ponteiro para a estrutura FILE anteriormente declarado;
nomearq: é o nome do arquivo a ser aberto e pode conter a especificação do drive e do diretório; pode ser um string literal entre aspas duplas ou um ponteiro para um string armazenado na memória;
modo: especifica o modo em que o arquivo deve ser aberto, ou seja, se o arquivo estará em modo *texto* ou *binário* e se será

usado para *leitura* e/ou *gravação* de dados, conforme detalhado a seguir.

Modos de abertura de arquivos:

Modo texto	Modo binário	Significado
r	rb	Abre o arquivo para leitura; se não existir, retorna *null*.
w	wb	Abre o arquivo para gravação; se não existir, cria-o; se já existir, apaga os dados já gravados.
a	ab	Abre o arquivo para acréscimo de dados (*append*); se não existir, cria-o; se já existir, acrescenta novos dados ao fim do arquivo.
r+	rb+	Abre o arquivo para leitura e gravação; se não existir, cria-o; se já existir, acrescenta os novos dados no início deste arquivo, sobregravando os já existentes.
w+	wb+	Abre o arquivo para leitura e gravação; se não existir, cria-o; se já existir, apaga os dados já existentes, sobregravando-os.
a+	ab+	Abre o arquivo para leitura e acréscimo de dados; se não existir, cria-o; se já existir, acrescenta os novos dados no fim do arquivo.

3.3 FECHANDO ARQUIVOS:

- **Ao encerrar o processamento do arquivo, devemos "fechá-lo", isto é, indicar ao sistema operacional que o arquivo não está mais sendo utlizado.**
- **Para fechar um arquivo usa-se o seguinte procedimento:**

```
fclose(fptr);
```

onde
fptr: é o ponteiro para a estrutura FILE que foi usado para referenciar o arquivo na abertura.

3.4 GRAVANDO ARQUIVOS, CARACTER A CARACTER:

- **Após abrir o arquivo para gravação, pode-se escrever nele usando a função** putc( )**, como no exemplo abaixo:**

```
/* Primeiro, abre-se o arquivo p/ gravação */
fptr = fopen("C:\\TESTE.TXT","w");

/* Mais adiante,pode-se efetuar a gravação */
putc('A',fptr);
```

Exemplo de gravação de arquivo em modo texto:

```
01: /* PROGRAMA EXEMPLO PARA GRAVAÇÃO DE UM ARQUIVO
02:    TEXTO, CARACTER A CARACTER */
03:
04: #include <stdio.h>
05: #include <conio.h>
06:
07: main()
08: {
09:    FILE *fptr;
10:    char ch;
11:
12:    /* Abre o arquivo especialmente para gravação */
13:    fptr = fopen("C:\\LUCIANA\\TESTE.TXT","w");
14:
15:    /* Lê um caracter do teclado até ser digitado um
16:       'enter' (return) usando a função getche()  */
17:
18:    while ((ch = getche()) != '\r')
19:        putc(ch,fptr);              // grava o caracter
20:
21:    /* Fecha o arquivo */
22:    fclose(fptr);
23: }
```

3.5 LENDO ARQUIVOS, CARACTER A CARACTER:

- **Para ler dados anteriormente gravados num arquivo é preciso, primeiramente, abri-lo para leitura; em seguida, pode-se ler caracter a caracter, até encontrar o caracter de *fim de arquivo* (EOF). Para esta forma de leitura, pode-se usar a função** getc( ), **como no exemplo abaixo:**

```
/* Primeiro, abre-se o arquivo p/ leitura */
fptr = fopen("C:\\TESTE.TXT","r");

/* Mais adiante,pode-se efetuar a leitura */
getc(fptr);
```

Exemplo de leitura de arquivo em modo texto:

```
01: /* PROGRAMA EXEMPLO PARA LEITURA DE UM ARQUIVO
02:    TEXTO, CARACTER A CARACTER */
03:
04: #include <stdio.h>
05:
06: main()
07: {
08:    FILE *fptr;
09:    int  ch;
10:
11:    /* Abre o arquivo especialmente para leitura */
12:    fptr = fopen("C:\\LUCIANA\\TESTE.TXT","r");
13:
14:    /* Lê um caracter do arquivo até ser encontrado um
15:     caracter de 'fim de arquivo' (EOF) com getc()  */
16:
17:    while ((ch = getc(fptr)) != EOF )
18:        putchar(ch);      // imprime o caracter na tela
19:
20:    /* Fecha o arquivo */
21:    fclose(fptr);
22: }
```

3.6 ERROS AO ABRIR ARQUIVOS:

- **Uma tentativa de abertura de arquivo pode resultar num erro.**

- **Por execmplo, você pode tentar abrir paa leitura um arquivo inexistente; pode não haver espaço em disco para gravação dos dados; pode ocorrer uma fallha na leitura do disco, etc.**

- **Por estes motivos, é importante que os programa que acessam arquivos em disco verifiquem se estes foram abertos com sucesso antes de tentar ler ou gravar informações.**

- **Para saber se houve um erro na abertura do arquivo, basta verificar se a função *fopen* retornou um caracter NULL:**

```
/* Checagem na abertura do arquivo p/ leitura */
if ((fptr = fopen("C:\\TESTE.TXT","r")) == NULL)
   puts("Erro ao abrir o arquivo");

/* Checagem na abertura do arquivo p/ gravação */
if ((fptr = fopen("C:\\TESTE.TXT","w")) == NULL)
   puts("Erro ao abrir o arquivo");
```

3.7 GRAVANDO ARQUIVOS, LINHA A LINHA:

- **É possível gravar strings diretamente num arquivo aberto para gravação através da função *fputs( )*. Para gerar um arquivo texto corretamente é necessário adicionar um caracter de "fim de linha" (\n) ao final de cada string.**

Exemplo de gravação de strings em arquivo em modo texto:

```
01: /* PROGRAMA EXEMPLO PARA GRAVAÇÃO DE UM ARQUIVO TEXTO,
02:    LINHA A LINHA */
03:
04: #include <stdio.h>
05: #include <string.h>
06:
07: main()
08: {
09:    FILE *fptr;
10:    char linha[80];
11:
12:   if ((fptr = fopen("C:\\TESTE.TXT","w")) == NULL)
13:       puts("Erro na abertura do arquivo");
14:    else
15:    {
16:       do
17:       {
18:          puts("Digite até 80 carateres para gravação no
                    arquivo (deixe em branco para sair):");
19:          gets(linha);
20:          if (strlen(linha)>0)   /* se digitou algo */
21:          {
22:             fputs(linha,fptr); // grava string no arq.
23:                 fputs("\n",fptr); // grava um 'fim de
linha'
24:          }
25:       } while (strlen(linha)>0);
26:    }
27:
28:    fclose(fptr);
29: }
```

3.8 LENDO ARQUIVOS, LINHA A LINHA:

- **Também é possível ler dados de um arquivo texto armazenando-os em strings através da função *fgets( )*.**
- **A função *fgets( )* deve ser acompanhada de um valor que indica o número máximo de caracteres a serem lidos de cada vez.**

Exemplo de leitura de strings em arquivo em modo texto:

```
01: /* PROGRAMA EXEMPLO PARA LEITURA DE UM ARQUIVO TEXTO,
02:  LINHA A LINHA */
03:
04: #include <stdio.h>
05: #include <string.h>
06:
07: main()
08: {
09:     FILE *fptr;
10:     char linha[80];
11:
12:    if ((fptr = fopen("C:\\TESTE.TXT","r")) == NULL)
13:       puts("Erro na abertura do arquivo");
14:    else
15:    {
16:       while (fgets(linha,80,fptr) != NULL)
17:          printf("%s",linha);
18:    }
19:
20:    fclose(fptr);
21: }
```

4. ARQUIVOS BINÁRIOS

4.1 LENDO E GRAVANDO REGISTROS:

- **A entrada e saída direta de arquivos é mais utlilizada para salvar dados que serão lidos posteriormente pelo mesmo programa ou por outro programa escrito em C.**

- **Esse modo de gravação/leitura só é usado em MODO BINÁRIO;**

- **As funções de entrada e saída direta são, respectivamente, *fread( )* e *fwrite( )*, que serão detalhadas a seguir.**

- **Em arquivos binários, pode-se ainda efetuar acesso aleatório aos arquivos usando a função *fseek( )*, como será visto mais adiante.**

4.2 GRAVAÇÃO DIRETA COM *fwrite( )*:

- **A função *fwrite( )* envia um bloco de dados da memória para um arquivo em modo binário, e é usada como abaixo:**

```
fwrite(endereço,tamanho,quantidade,ponteiro);
```

onde:

- endereço **é o endereço de memória onde se encontram os dados que devem ser gravados no arquivo (é um ponteiro do tipo *void*);**
- tamanho **corresponde ao número de bytes de cada item individual da dados que será gravado;**
- quantidade **especifica quantos itens individuais serão**

gravados simultaneamente;

- ponteiro **é o nome do ponteiro usado para indicar o arquivo conforme especificado na abertura do mesmo;**

- **A função *fwrite( )* retorna o número de itens gravados com sucesso; se este número <u>for menor que</u> tamanho, isso significa que <u>ocorreu algum erro</u>.**

<u>Exemplos do uso da função fwrite:</u>

```
/* Grava uma única variável do tipo int */
fwrite(&x,sizeof(int),1,fptr);

/* Grava uma única variável do tipo double */
fwrite(&y,sizeof(float),1,fptr);

/* Grava uma matriz denominada aluno contendo 50
   estruturas do tipo registro */
fwrite(aluno,sizeof(registro),50,fptr);

/* Grava a matriz anterior como um único bloco */
fwrite(matriz,sizeof(matriz),1,fptr);

/* Grava uma variável do tipo int e testa o sucesso*/
if ( fwrite(&x,sizeof(int),1,fptr) =! 1) ...// erro
```

Programa exemplo com gravação direta (modo binário):

```
01: /* PROGRAMA EXEMPLO DE GRAVAÇÃO DIRETA EM ARQUIVO BINÁRIO */
02: #include <stdio.h>
03: #include <string.h>
04:
05:
06: struct data
07: {
08:    int dia;
09:    int mes;
10:    int ano;
11: };
12:
13: struct registro
14: {
15:    char nome[20];
16:    char endereco[30];
17:    char fone[15];
18:    char cpf[11];
19:    struct data nasc;
20: } cliente;
21:
22: main()
23: {
24:    FILE *fptr;
25:
26:    fptr = fopen("C:\\CLIENTE.BIN","wb");
27:
28:    do
29:    {
30:       fflush(stdin);
31:       puts("Digite o nome do cliente ou <FIM> para sair:");
32:       gets(cliente.nome);
33:       if ( strcmp ( cliente.nome, "FIM" ) != 0 )
34:       {
35:          puts("Digite o endereço do cliente:");
36:          gets(cliente.endereco);
37:          puts("Digite o telefone do cliente:");
38:          gets(cliente.fone);
39:          puts("Digite o CPF do cliente:");
40:          gets(cliente.cpf);
41:          puts("Digite dia, mes e ano de nasc. do cliente:");
42:          scanf("%d %d %d",&cliente.nasc.dia,
                      &cliente.nasc.mes,&cliente.nasc.ano);
43:          fwrite(&cliente,sizeof(registro),1,fptr);
44:       }
45:    } while ( strcmp ( cliente.nome, "FIM" ) != 0 );
46:    fclose(fptr);
47: }
```

4.3 LEITURA DIRETA COM *fread( )*:

- **A função *fread( )* lê um bloco de dados de um arquivo em modo binário para a memória e é usada como abaixo:**

```
fread(endereço,tamanho,quantidade,ponteiro);
```

onde:

- endereço **é o endereço de memória onde os dados lidos serão armazenados (é um ponteiro do tipo *void*);**
- tamanho **corresponde ao número de bytes de cada item individual da dados que será lido;**
- quantidade **especifica quantos itens individuais serão lidos simultaneamente;**
- ponteiro **é o nome do ponteiro usado para indicar o arquivo conforme especificado na abertura do mesmo;**

- **A função *fread( )* retorna o número de itens lidos que pode ser menor que** tamanho **se for lido um caractere de *fim-de-arquivo* ou se ocorrer algum erro na leitura.**

Exemplos do uso da função fwrite:

```
/* Lê uma única variável do tipo int */
fread(&x,sizeof(int),1,fptr);

/* Lê uma única variável do tipo double */
fread(&y,sizeof(float),1,fptr);

/* Lê dados do arquivo e armazena-os numa matriz denominada
   aluno contendo 50 estruturas do tipo registro */
fread(aluno,sizeof(registro),50,fptr);

/* Lê dados do arquivo e armazena-os na matriz anterior
   de uma só vez (como um único bloco) */
fread(matriz,sizeof(matriz),1,fptr);

/* Lê uma variável do tipo int e testa o sucesso*/
if ( fread(&x,sizeof(int),1,fptr) =! 1) ...// erro
```

Programa exemplo com leitura direta (modo binário):

```
01: /* PROGRAMA EXEMPLO DE LEITURA DIRETA EM ARQUIVO BINÁRIO */
02: #include <stdio.h>
03: #include <string.h>
04:
05: struct data
06: {
07:    int dia;
08:    int mes;
09:    int ano;
10: };
11:
12: struct registro
13: {
14:    char nome[20];
15:    char endereco[30];
16:    char fone[15];
17:    char cpf[11];
18:    struct data nasc;
19: } cliente;
20:
21: main()
22: {
23:    FILE *fptr;
24:
25:    fptr = fopen("C:\\CLIENTE.BIN","rb");
26:
27:    do
28:    {
29:       /* se leitura for feita com sucesso, imprima dados */
30:       if ( fread(&cliente,sizeof(registro),1,fptr) == 1 )
31:       {
32:          printf("%s %s %s %s %d/%d/%d\n",cliente.nome,
           cliente.endereco,cliente.fone,cliente.cpf,

cliente.nasc.dia,cliente.nasc.mes,cliente.nasc.ano);
33:       }
34:    } while (! feof(fptr) );
35:
36:    fclose(fptr);
37: }
```

4.4 TIPOS DE ACESSO A ARQUIVOS:

- **Todo arquivo aberto tem a ele associado um "ponteiro de arquivo" que aponta para a posição atual do arquivo, indicando qual o próximo byte que deve ser lido e/ou gravado;**

- **Esse "ponteiro" ou "indicador de posição" é especificado em termos de bytes a partir da posição inicial do arquivo, que é a posição 0 (zero);**

- **Quando um arquivo é novo, seu comprimento é zero e não há outra posição a ser indicada;**

4.4.1 ACESSO SEQÜENCIAL

- **Nos exemplos vistos até agora, os arquivos eram acessados de forma seqüencial, com as seguintes características:**
  - **Quando um arquivo já existe e é aberto, o "indicador de posição" aponta para o final do arquivo caso tenha sido especificado o modo de acréscimo de dados (*append*), ou para o seu início quando outro modo é usado;**
  - **Quando é feita a leitura de dados, o "indicador de posição" é automaticamente movido para a próxima posição a ser lida, seqüencialmente.**
  - **O mesmo ocorre durante a gravação, ou seja, após gravar os novos dados logo abaixo dos dados existentes, o "indicador de posição" é automaticamente movido para a posição subseqüente (seqüencialmente);**

4.4.2 CONTROLANDO O "INDICADOR DE POSIÇÃO"

- **A função *rewind( )* permite mover o "indicador de posição" do para o início do arquivo, independentemente de onde se encontre num dado momento;**

- **A sintaxe desta função é:**

```
rewind(fptr)
```

onde

- fptr **é o nome do ponteiro usado para indicar o arquivo conforme especificado na abertura do mesmo;**

- **A função *ftell( )* indica o valor atual do "indicador de posição" de um arquivo.**

- **Esta função retorna um inteiro longo (que é o número do byte onde está localizado o "indicador de posição") e sua sintaxe é:**

```
ftell(fptr)
```

onde

- fptr **é o nome do ponteiro usado para indicar o arquivo conforme especificado na abertura do mesmo;**

4.4.3 ACESSO ALEATÓRIO

- **O acesso aleatório, ao contrário do seqüencial, permite que a transferência de dados seja feita de/para qualquer posição do arquivo, sem que as informações anteriores precisem ser acessadas.**
- **A função que permite o acesso aleatório a um arquivo é *fseek( )* e sua sintaxe é a seguinte:**

```
fseek(fptr,offset,origem)
```

onde

- fptr **é o nome do ponteiro usado para indicar o arquivo conforme especificado na abertura do mesmo;**
- offset **é o deslocamento, em bytes, contado a partir da *origem;***
- origem **indica a origem do deslocamento e pode ser:**
  - **0 (início do arquivo)**
  - **1 (posição atual do "indicador de posição")**
  - **2 (fim do arquivo)**

Note que, quando a origem especificado for o "fim do arquivo" (valor 2), o deslocamento (offset) deve ser negativo;

5. Funções de gerenciamento de arquivos:

5.1 Apagando um arquivo:

```
/* Demonstra o uso da função remove().*/
#include <stdio.h>
#include <stdlib.h>

main()
{
  char nomearq[80];

  printf("Digite o nome do arquivo a ser apagado:");
  gets(nomearq);

  if(remove(nomearq)==0)
    printf("O arquivo %s foi apagado.",nomearq);
  else
    printf("Erro ao apagar o arquivo %s.",nomearq);
}
```

5.2 Mudando o nome de um arquivo:

```
/* Mudando o nome de um arquivo usando rename().*/
#include <stdio.h>
#include <stdlib.h>

main(){

  char oldname[80], newname[80];

  printf("Digite o nome atual:");
  gets(oldname);
  printf("Digite o novo nome:");
  gets(newname);

  if (rename(oldname, newname) == 0)
     printf("O nome de %s foi mudado para %s.",
             oldname, newname);
  else
     printf("Erro ao renomear %s.",oldname);
 }
```

5.3 Copiando um arquivo:

```
/* Copiando um arquivo .*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int copia_arq(char *nomeorigem, char *nomedestino);

main(){

  char origem[80], destino[80];

  /* Obtem os nomes dos arquivos de origem e de destino.*/

  printf("\nDigite o nome do arquivo de origem:");
  gets(origem);
  printf("\nDigite o nome do arquivo de destino:");
  gets(destino);

  if(copia_arq(origem, destino) == 0)
    puts("Arquivo copiado sem problemas.");
  else
    printf("Erro durante a operação de cópia.");
}

int copia_arq(char *origem, char *destino){

  FILE *forigem, *fdestino;
  int c;

  /*Abre o arq.de origem para leitura em modo binário.*/

  if ((forigem = fopen(origem, "rb"))== NULL)
      return -1;

  /*Abre arq.de destino para gravação em modo binário.*/

  if ((fdestino = fopen(destino, "wb"))== NULL){

    fclose(forigem);
    return -1;
  }

  /* Le 1 byte de cada vez do arq. de origem.   */
```

```
  /* se o fim do arquivo ainda não foi atingido, */
  /* grava o byte no arquivo de destino.*/

  while(1)
  {
    c= fgetc(forigem);
    if (!feof(forigem))
       fputc(c, fdestino);
    else
       break;
  }
  fclose(forigem);
  fclose(fdestino);

  return 0;
}
```

5.4 Usando arquivos temporários:

```
/* Demonstra o uso de arquivos temporarios.*/

#include <stdio.h>
main(){

  char buffer[10], *c;

  /*Obtem um nome temporario no buffer especificado.*/

  tmpnam(buffer);

  /*Obtem outro nome, desta vez no buffer interno*/
  /* da própria função.*/

  c = tmpnam(NULL);

  /* Exibe os nome.*/

  printf("O nome temporario 1 e $\%s", buffer);
  printf("\nO nome temporario 2 e %s", c);
}
```