LINGUAGEM DE PROGRAMAÇÃO C

Prof. Marco Aurélio da Fonseca

Apostila #2

1. Matrizes

Correspondem a elementos do mesmo tipo, agrupados sob o mesmo nome e diferenciados entre si através de índices. Na linguagem C, todas as matrizes consistem em posições contíguas, sendo que o endereço de memória mais baixo corresponde ao primeiro elemento e o endereço mais alto ao último elemento.

Os valores armazenados na matriz são chamados de "elementos da matriz". O primeiro elemento da matriz é indexado como índice zero e o último é indexado como total de elementos menos 1. Assim, para uma matriz “int nota[30]”, o primeiro elemento é nota[0] e o último elemento é nota[29].

Ex:

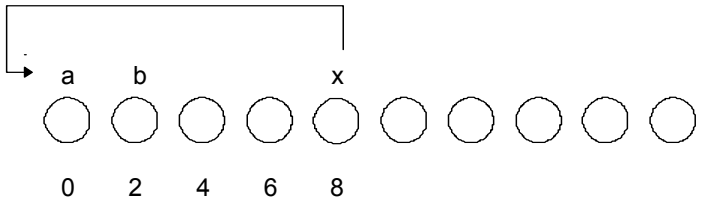
```
void main()
{
char nome[20];
float preco[30];
int m[5][3]; /* bidimensional */
char c[3] = {’f’,’i’,’m’}; /* declarada e inicializada */
char d[3] = “fim”; /* idem */
int a[5] = {1,10,3,5,30} /* declarada e inicializada, numérica */
}
```

Ex 2: Imprima 5 números na ordem inversa a que forem informados.

```
void main()
{
int i,a[5];
for (i=0;i<=4;i++) {
        printf(“Elemento %d: “,i);
        scanf(“%d”,a[i]);
}
for (i=4;i>=0;i--)
        printf(“Elemento %d: “,a[i]);
}
```

2. Ponteiros

Ponteiros são endereços, isto é, são variáveis que contém um endereço de memória. Se uma variável contém o endereço de outra, então a primeira (o ponteiro) aponta para a segunda.



"x" o "ponteiro" aponta para o "inteiro" a

2.1. Operadores para ponteiros

& (E comercial) que fornece o endereço de determinada variável. Não confundir com o operador lógico de operações de baixo nível, de mesmo símbolo. Atribui o endereço de uma variável para um ponteiro.

* (Asterístico) que acessa o conteúdo de uma variável, cujo endereço é o valor do ponteiro. Não confundir com o operador aritmético de multiplicação de mesmo símbolo. Devolve o valor endereçado pelo ponteiro.

Ex:

```
void main()
{
int *pont, cont, valor;
cont = 100;
pont = &cont;
val = *pont;
printf("%d",val);   /* 100 */
}
```

2.2. Aritmética com Ponteiros

São válidas as operações de soma e subtração, sendo que seu resultado depende do tipo de variável apontada pelo ponteiro.

Supondo que:

```
int *p, x;
char *q, a;
q = &a;
p = &x;
```

E ainda que:

a = endereço 100

x = endereços 101/102

Então:

q++ --> q “apontará” para o endereço 101, pois variáveis do tipo char ocupam 1 byte na memória

p++ --> p “apontará” para o endereço 103, pois variáveis do tipo int ocupam 2 bytes na memória

Este conceito é particularmente importante no que se refere a matrizes pois como se sabe, matriz nada mais é que um conjunto de variáveis do mesmo tipo, dispostas seqüencialmente em memória.

Ex:

```
void main()
{
int x,y,*px,*py;
x = 100;
px = &x;   /* px tem o endereco de x */
py = px;   /* py tem o endereco de x */
y = *py;   /* y vale 100, pois recebe o conteúdo de x, através do  ponteiro py */
printf(“%d %d”,x,y);
}
```

3. Estruturas

Em C, uma estrutura é uma coleção de variáveis referenciadas através de um nome definido pelo programador. Exemplo de estrutura para um pequeno programa de cadastro de clientes:

```
struct dados {
        char nome[30];
        char rua[40];
        char cidade[20];
        char estado[02];
        char cep[09];
};
```

Não devemos confundir tipo com variável, de forma que é errado afirmar-se coisas como “a variável dados recebeu ...”, este tipo (dados) servirá para posteriormente declaramos uma variável como segue:

```
struct dados cliente;
```

Ou ainda:

```
struct dados {
        char nome[30];
        char rua[40];
        char cidade[20];
        char estado[02];
        char cep[09];
} cliente;
```

O exemplo a seguir mostra a utilização de uma estrutura em um programa de cadastro, que guarda as informações em um arquivo de dados. As funções de manipulação de arquivos serão descritas no próximo capítulo.

```
struct registro {
    char nome[40];
    char endereco[40];
    float valor;
} matriz[100];

int saida()
{
clrscr();
exit(0);
}

void inicia_matriz()
{
int t;
for (t=0;t<100;t++)
    *matriz[t].nome = '\0';
}
```

```
char menu()
{
char s;
clrscr();
do {
    puts("Inserir");
    puts("Exibir");
    puts("Carregar");
    puts("Salvar");
    puts("Finaliza");
    printf("Digite a 1a. Letra: ");
    scanf("%c",&s);
} while(s != 'i' && s != 'e' && s != 'c' && s != 's' && s != 'f');
return(s);
}

void inserir()
{
int i;
for (i=0; i < 100; i++)
    if (!*matriz[i].nome) break;
      if (i==100) {
          puts("Arquivo Cheio!");
          return;
      }
      printf("Nome: "); gets(matriz[i].nome);
      printf("End.: "); gets(matriz[i].endereco);
      printf("Val.: "); scanf("%f",&matriz[i].valor);
}

void exibir()
{
char x;
int t;
```

```
clrscr();
for(t=0;t<100;t++) {
    if (*matriz[t].nome) {
      printf("%s \n", matriz[t].nome);
      printf("%s \n", matriz[t].endereco);
      printf("%f \n", matriz[t].valor);
      puts(" "); puts("<Enter> para prosseguir!");
      x = getchar();
      }
    else
      break;
    }
}

int salvar()
{
FILE *fp;
int i;
if ((fp=fopen("LISTA.DAT","wb"))==NULL) {
    puts("Falhou Abertura! ");
    return;
}
for (i=0;i<100;i++)
    if (*matriz[i].nome)
      if (fwrite(&matriz[i],sizeof(struct registro), 1,fp) != 1)
        puts("Falha na Gravacao! ");
    fclose(fp);
}

int carga()
{
FILE *fp;
int i;
if ((fp=fopen("LISTA.DAT","rb")) == NULL) {
    puts("Falha na Abertura do Arquivo!");
```

```
    return;
}
inicia_matriz();
for (i=0; i < 100; i++)
    if (fread(&matriz[i], sizeof(struct registro), 1, fp) != 1) {
      if (feof(fp)) {
        fclose(fp);
        return;
         }
    else {
      puts("Erro de Leitura! ");
      fclose(fp);
      return;
     }
    }
}

void main()
{
char escolha;
inicia_matriz();
for (;;) {
    escolha = menu();
    switch (escolha) {
      case 'i' :
        inserir();
        break;
      case 'e' :
        exibir();
        break;
      case 'c' :
        carga();
        break;
      case 's' :
        salvar();
```

```
        break;
      case 'f' :
        saida();
       }
   }
}
```

4. Alocação Dinâmica de Memória

A alocação dinâmica permite ao programador alocar memória para variáveis quando o programa está sendo executado. Assim, poderemos definir, por exemplo, um vetor ou uma matriz cujo tamanho descobriremos em tempo de execução. O padrão C define algumas funções para o sistema de alocação dinâmica:

- malloc
- calloc
- free

No entanto, existem diversas outras funções que são amplamente utilizadas, mas dependentes do ambiente e compilador. Neste curso serão abordadas somente estas funções padronizadas.

4.1. malloc

A função malloc() serve para alocar memória e tem o seguinte protótipo:

```
void *malloc (unsigned int num);
```

A função toma o número de bytes que queremos alocar (**num**), aloca na memória e retorna um ponteiro **void *** para o primeiro byte alocado. O ponteiro **void *** pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função **malloc()** retorna um ponteiro nulo.

Ex:

```
void main ()
{
int *p;
int a;
int i;
p= malloc(a*sizeof(int));
```

```
if (!p)
{
        printf ("** Erro: Memoria Insuficiente **");
        exit;
}
for (i=0; i<a ; i++)
{
        p[i] = i*i;
}
}
```

No exemplo acima, é alocada memória suficiente para se armazenar a números inteiros. O operador sizeof() retorna o número de bytes de um inteiro. Ele é útil para se saber o tamanho de tipos. O ponteiro void* que malloc() retorna é convertido para um int* pelo cast e é atribuído a p. A declaração seguinte testa se a operação foi bem sucedida. Se não tiver sido, p terá um valor nulo, o que fará com que !p retorne verdadeiro. Se a operação tiver sido bem sucedida, podemos usar o vetor de inteiros alocados normalmente, por exemplo, indexando-o de p[0] a p[(a-1)].

4.2. calloc

A função calloc() também serve para alocar memória, mas possui um protótipo um pouco diferente:

```
void *calloc (unsigned int num, unsigned int size);
```

A função aloca uma quantidade de memória igual a num * size, isto é, aloca memória suficiente para um vetor de num objetos de tamanho size. Retorna um ponteiro void * para o primeiro byte alocado. O ponteiro void * pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função calloc() retorna um ponteiro nulo. Em relação a malloc, calloc tem uma diferença (além do fato de ter protótipo diferente): calloc inicializa o espaço alocado com 0.

Ex:

```
void main ()
{
int *p;
int a;
int i;
```

```
p=  calloc(a,sizeof(int));
if (!p)
{
        printf ("** Erro: Memoria Insuficiente **");
        exit;
}
for (i=0; i<a ; i++)
{
        p[i] = i*i;
}
}
```

No exemplo acima, é alocada memória suficiente para se colocar a números inteiros. O operador sizeof() retorna o número de bytes de um inteiro. Ele é útil para se saber o tamanho de tipos. O ponteiro void * que calloc() retorna é convertido para um int * pelo cast e é atribuído a p. A declaração seguinte testa se a operação foi bem sucedida. Se não tiver sido, p terá um valor nulo, o que fará com que !p retorne verdadeiro. Se a operação tiver sido bem sucedida, podemos usar o vetor de inteiros alocados normalmente, por exemplo, indexando-o de p[0] a p[(a-1)].

4.3. free

Quando alocamos memória dinamicamente é necessário que nós a liberemos quando ela não for mais necessária. Para isto existe a função free() cujo protótipo é:

```
void free (void *p);
```

Basta então passar para free() o ponteiro que aponta para o início da memória alocada. Mas você pode se perguntar: como é que o programa vai saber quantos bytes devem ser liberados? Ele sabe pois quando você alocou a memória, ele guardou o número de bytes alocados numa "tabela de alocação" interna.

Ex:

```
void main ()
{
int *p;
int a;
p= malloc(a*sizeof(int));
```

```
if (!p)
{
      printf ("** Erro: Memoria Insuficiente **");
      exit;
}
free(p);
}
```

5. Entradas e Saídas em Arquivos

As principais funções para manipulação de arquivos em C são:

Nome	Função
fopen()	Abre uma fila
fclose()	Fecha uma fila
feof()	Devolve se fim de fila
ferror()	Devolve Verdadeiro se um erro tiver ocorrido
fprint()	Saída
Fscanf()	Entrada
Fseek()	Procura um byte especificado na fila
getc()	Lê um caracter na fila
putc()	Grava um caracter na fila
remove()	Apaga o arquivo
rewind()	Reposiciona o ponteiro do Arquivo em seu início

A seguir descreveremos estas funções exemplificando seu uso.

5.1. fopen

A função fopen tem duas finalidades, a saber:

1. abrir uma fila de bytes
2. ligar um arquivo em disco àquela fila

Sintaxe:

```
FILE *fopen (char *NomeArquivo, char *modo);
```

A tabela a seguir apresenta os modos de abertura de arquivo válidos:

Modo	Significado
“r”	Abre Arquivo de Texto para Leitura
“w”	Cria Arquivo de Texto para Gravação

“a”	Anexa a um Arquivo de Texto
“rb”	Abre Arquivo Binário para Leitura
“wb”	Cria Arquivo Binário para Gravação
“ab”	Anexa a um Arquivo Binário
“r+”	Abre Arquivo de Texto para Leitura/Gravação
“w+”	Cria Arquivo de Texto para Leitura/Gravação
“a+”	Abre ou Cria Arquivo de Texto para Leitura/Gravação
“r+b”	Abre Arquivo Binário para Leitura/Gravação
“w+b”	Cria Arquivo Binário para Leitura/Gravação
“a+b”	Abre ou Cria Arquivo Binário para Leitura/Gravação
“rt”	Idem a “r”
“wt”	Idem a “w”
“at”	Idem a “a”
“r+t”	Idem a “r+”
“w+t”	Idem a “w+”
“a+t”	Idem a “a+”

No caso de falha na abertura do arquivo, a função retorna um ponteiro do tipo NULL.

5.2. fclose

Fecha uma fila. Caso o programa seja encerrado sem que as filas sejam fechadas, dados gravados nos buffers podem ser perdidos. Sintaxe:

```
fclose (FILE *fp);
```

Ex:

```
void main()
{
FILE *fp;
fp = fopen(“teste.txt”,”rb”);
if (!fp) {
        printf(“Arquivo não pode ser aberto\n”);
        exit(1);
        }
fclose(fp);
}
```

5.3. ferror

Determina se a operação de arquivo produziu um erro. Sua forma geral será:

```
int ferror(FILE *fp);
```

5.4. fseek

Posiciona o ponteiro de leitura/gravação de arquivo em uma determinada posição em arquivos abertos para acesso aleatório. Sintaxe:

```
int fseek(FILE *fp, long int num_bytes, int origem);
```

Onde:

fp - é o ponteiro de arquivo devolvido por fopen().

num_bytes - é um inteiro longo que representa o número de bytes desde a origem até chegar a posição corrente.

Este comando é normalmente utilizado em arquivos binários.

Ex:

```
void main()
{
FILE *fp;
fp = fopen("teste","rb");
if (!fp)
{
        printf("Arquivo não pode ser aberto\n");
        exit(1);
}
fseek(fp, 234L, 0);      /* L força que seja um inteiro longo */
getc(fp);       /* lê o caracter 234 */
}
```

5.5. rewind

Reinicia o arquivo, ou seja, apenas movimenta o ponteiro do arquivo para seu início. Sintaxe:

```
rewind(FILE *fp);
```

5.6. feof

EOF ("End of file") indica o fim de um arquivo. Às vezes, é necessário verificar se um arquivo chegou ao fim. Para isto podemos usar a função feof(). Ela retorna não-zero se o arquivo chegou ao EOF, caso contrário retorna zero. Seu protótipo é:

```
int feof (FILE *fp);
```

Outra forma de se verificar se o final do arquivo foi atingido é comparar o caractere lido por getc com EOF. O programa a seguir abre um arquivo já existente e o lê, caracter por caracter, até que o final do arquivo seja atingido. Os caracteres lidos são apresentados na tela.

```
void main()
{
FILE *fp;
char c;
fp = fopen("arquivo.txt","r");   /* Arquivo ASCII, para leitura */
if(!fp)
{
   printf( "Erro na abertura do arquivo");
   exit(0);
}
c = getc(fp);
while(c != EOF)      /* Enquanto não chegar ao final do arquivo */
{
   printf("%c", c);              /* imprime o caracter lido */
}
fclose(fp);
}
```

5.7. remove

Apaga um arquivo especificado. Sintaxe:

```
int remove (char *nome_do_arquivo);
```

Ex:

```
void main()
{
char str[255];
```

```
/* Le um nome para o arquivo a ser aberto: */
printf("\n\n Entre com o nome do arquivo a apagar:\n");
gets(str);
remove(str);
}
```

5.8. putc

Grava um caracter em fila previamente aberta. Sintaxe:

```
int putc(int ch, FILE *fp);
```

Onde:

ch - é o caracter a ser gravado

fp - é o ponteiro devolvido por fopen

5.9. getc

Lê caracter em uma fila aberta. Sintaxe:

```
int getc(FILE *fp);
```

Ex:

```
void main()
{
FILE *p;
p = fopen(str,"r");
if (!p)
{
        printf("Erro! Impossivel abrir o arquivo!\n");
        exit(1);
 }
ch = getc(fp);
while (ch != EOF)
{
        ch = getc(fp);
}
}
```

5.10. fprintf

A função fprintf() funciona como a função printf(). A diferença é que a saída de fprintf() é um arquivo e não a tela do computador. Protótipo:

```
int fprintf (FILE *fp,char *str,...);
```

Como já poderíamos esperar, a única diferença do protótipo de fprintf() para o de printf() é a especificação do arquivo destino através do ponteiro de arquivo.

5.11. fscanf

A função fscanf() funciona como a função scanf(). A diferença é que fscanf() lê de um arquivo e não do teclado do computador. Protótipo:

```
int fscanf (FILE *fp,char *str,...);
```

Como já poderíamos esperar, a única diferença do protótipo de fscanf() para o de scanf() é a especificação do arquivo destino através do ponteiro de arquivo.

Ex:

```
void main()
{
FILE *p;
char str[80],c;
/* Le um nome para o arquivo a ser aberto: */
printf("\n\n Entre com um nome para o arquivo:\n");
gets(str);
p = fopen(str,"w");
if (!p)  /* Caso ocorra algum erro na abertura do arquivo..*/
{
        printf("Erro! Impossivel abrir o arquivo!\n");
        exit(1);
 }
/* Se nao houve erro, imprime no arquivo, fecha ...*/
fprintf(p,"Este e um arquivo chamado:\n%s\n", str);
fclose(p);
p = fopen(str,"r");
while (!feof(p))
```

```
{
        fscanf(p,"%c",&c);
        printf("%c",c);
}
fclose(p);
}
```

5.12. fgets e fputs

Sintaxes:

```
char *fputs(char *str, FILE *fp);
char *fgets(char *str, int comprimento, FILE *fp);
```

fputs() é análoga a puts(), porém escreve em disco.

fgets() lê uma “string” da fila especificada, incluindo caracteres como \n, porém a “string” lida sempre será finalizada com zero ASCII (\0).

5.13. fread e fwrite

Permitem que leiamos/gravemos blocos de dados, sua forma geral é a seguinte:

```
int fread(void *buffer, int num_bytes, int cont, FILE *fp);
int fwrite(void *buffer, int num_bytes, int cont, FILE *fp);
```

Ex - Leitura de arquivos contendo números:

```
void main()
{
FILE *fp;
float f = 12.23;
fp=fopen(“teste”,”wb”);
if (!fp)
{
        printf(“Arquivo não pode ser criado\n”);
        exit(1);
}
fwrite(&f, sizeof(f), 1, fp);
```

```
fclose(fp);
}
```