



Universidade Estadual de Campinas – UNICAMP

Centro Superior de Educação Tecnológica – CESET

Linguagem C - vol. 2
Prof.: Marco Antonio Garcia de Carvalho

Setembro 2003
Campinas, SP - Brasil

Sumário

1	Matrizes	2
1.1	Declaração de matrizes	2
1.2	Inicialização de matrizes	2
1.3	Matrizes de mais de uma dimensão	2
1.4	Manipulação de strings	3
1.5	Exemplos de programas	3
1.6	Exercícios	3
2	Estruturas	6
2.1	Criando estruturas com <i>struct</i>	6
2.2	Declarando uma variável	6
2.3	Outras operações com estruturas	7
2.4	Typedef	7
2.5	Enumerações	8
2.6	Exemplos & exercícios	8
3	Funções	9
3.1	Introdução	9
3.2	Protótipo e definição de funções	9
3.3	Passagem de parâmetros	9
3.4	Funções recursivas	11
3.5	Exercícios	11
4	Arquivos	15
4.1	Introdução	15
4.2	Arquivo de acesso seqüencial	15
4.2.1	Criando arquivos	15
4.2.2	Gravando dados em um arquivo	16
4.2.3	Lendo dados de um arquivo	16
4.3	Exemplos mais elaborados	16
4.4	Exercícios	18
5	Alocação dinâmica	19
5.1	Introdução	19
5.2	Alocando memória dinamicamente para um vetor	19
	Bibliografia	20

1 Matrizes

É um tipo de dado usado para representar uma coleção de variáveis de mesmo tipo. São referenciadas por um único nome, diferenciadas por um ou mais índices. A palavra matriz aplica-se mais quando trabalhamos com dois índices (variável composta bidimensional). Vetores indicam uma variável composta unidimensional.

1.1 Declaração de matrizes

A declaração da variável é seguida por um par de colchetes contendo um número inteiro indicando o tamanho da matriz, podendo também conter uma constante declarada anteriormente.

```
Ex.:  
int notas[5]; // cinco elementos inteiros  
float corrente[2]; // 2 elementos reais
```

No exemplo acima, cada elemento é referenciado por um único índice, sendo estes **iniciados por zero**. Os elementos de uma matriz são armazenados em seqüência contínua de memória.

- **Obs.:** C não avisa quando o limite de uma matriz foi excedido. Se você ultrapassar o valor do limite imposto na declaração da matriz, os valores sobressalentes irão sobrepor outros dados da memória.

1.2 Inicialização de matrizes

A lista de valores é colocada entre chaves e separados por vírgulas. A inicialização deve ser concluída com um ponto e vírgula.

```
Ex.:  
int dmes[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

A instrução de definição de uma matriz inicializada pode suprimir a dimensão da matriz. Quando nenhum valor é fornecido, o compilador contará o número de valores inicializadores e o fixará como dimensão da matriz.

Ex.:

```
int dmes [] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

1.3 Matrizes de mais de uma dimensão

Utiliza-se dois pares de colchetes. Cada par de colchetes adicionais obtemos matrizes com uma dimensão a mais. A lista de valores usada para inicializar a matriz é uma lista de constantes separadas por vírgulas e envolta por chaves.

```
Ex.:  
int soma[3][3];  
float nota[2][3] = {{9, 8, 7}, {6.5, 4, 2}};
```

1.4 Manipulação de strings

Não existe na linguagem C um tipo de dado *string*: utiliza-se uma matriz de caracteres (tipo **char**, como visto abaixo) que termina com um caracter de controle '\0' (colocado pelo compilador).

```
char palavra[10]="exemplo";
```

Não se pode fazer em C, por exemplo:

```
stringA = stringB;
```

Strings devem ser igualadas elemento a elemento. A biblioteca padrão C possui diversas funções para trabalhar com strings.

Nome da função	Descrição
<code>gets(<i>palavra1</i>)</code>	Lê a string <i>palavra1</i> a partir do teclado
<code>strcpy(<i>palavra-destino</i>, <i>palavra-origem</i>)</code>	Copia a string <i>palavra-origem</i> para <i>palavra-destino</i>
<code>strcat(<i>palavra-destino</i>, <i>palavra-origem</i>)</code>	Anexa a string <i>palavra-origem</i> no final da string <i>palavra-destino</i>
<code>strlen(<i>palavra</i>)</code>	Retorna o comprimento da string <i>palavra</i>
<code>strcmp(<i>palavra1</i>, <i>palavra2</i>)</code>	Compara as strings <i>palavra1</i> e <i>palavra2</i> : se forem iguais, retorna zero.

Tabela 1: Principais funções de manipulação de strings. Com exceção de `gets()`, todas as funções desta tabela estão definidas em **string.h**.

1.5 Exemplos de programas

Nos exemplos a seguir, escreveremos somente a função `main()`.

- Ex.: Tabuada utilizando vetores.

```
// Tabuada
main()
{
  int result[11], i, t = 7;
  for(i = 0; i < 11; i++)
    result[i] = i * t;
  for(i = 0; i < 11; i++)
    printf ("%d * %d = %d \n", i, t, result[i]);
}
```

- Ex.: Pesquisa em vetores.
- Ex.: Matrizes - Leitura e escrita de valores.

1.6 Exercícios

1. (**intercalação de vetores**) Ler dois vetores de números reais *A* e *B* com 10 elementos cada. Construir um vetor *C* a partir da intercalação dos elementos de *A* e *B*, ou seja, $C_1 = A_1$, $C_2 = B_1$, $C_3 = A_2$, $C_4 = B_2 \dots$

```

// Pesquisa em vetores
main()
{
int vetorA [10] = {1, 28, 31, 30, 3, 0, 8, 12, 5, 21}, i, elem, chave = 0;
printf ("Digite um valor\n"); scanf("%d", &elem);
for(i = 0; i < 10; i++)
if(vetorA [i] == elem)
  chave = i+1;
if(chave != 0)
printf ("O elemento %d estah na posicao %d \n", elem , chave);
else
printf ("Elemento %d nao existe no vetor especificado \n", elem);
}

```

```

// Leitura e escrita de matrizes
main()
{
float matA [3] [3];
/* Entrada de dados na matriz */
printf ("Informe o valor de cada elemento —\n");
for(int i = 0; i < 3; i++)
for(int j = 0; j < 3; j++)
{
printf ("matA(%d,%d) = \n", i+1,j+1);
scanf("%f", &matA [i] [j]);
}
/* Impressao da matriz lida */
printf ("A matriz lida eh= \n");
for(int i = 0; i < 3; i++)
{
printf ("\n ");
for(int j = 0; j < 3; j++)
printf ("%f ", matA [i] [j]);
}
}
}

```

2. (**strings**) Leia uma string de até 10 caracteres. Em seguida, escreva-a colocando um espaço em branco entre cada caracter.
3. (**urna eletrônica**) Haverá uma eleição para representante de turma na disciplina de C, no CESET. Dois alunos se candidataram e a comissão eleitoral decidiu contar votos brancos e nulos. Elabore um programa em C para que a eleição seja totalmente informatizada. Em seguida, apresente um relatório final da comissão, com os votos de cada candidato e o resultado final da eleição.
4. (**inversão de strings**) Leia uma string de até 10 caracteres. Em seguida, escreva-a de forma invertida. Por exemplo, caso seja lida a palavra *Marco*, o programa deve retornar *ocraM*.
5. (**ordenação Bolha**) É uma das ordenações mais conhecidas (e de pior performance, da ordem de n^2). A ordenação envolve repetidas comparações e, se necessário, a troca de elementos adjacentes. Os elementos são como bolhas em um tanque de água: cada um procura o seu próprio nível. Dado o Algoritmo 1 abaixo, implemente-o em linguagem C para um vetor de números inteiros.

Algoritmo 1 Algoritmo *Bolha*

Entrada: Vetor V , número de elementos n de V

Saída: Vetor V ordenado

Para i de 2 até n **Faça**

Para j de n até i **Faça**

Se $V(j-1) > V(j)$ **Então**

$temp \leftarrow V(j-1)$

$V(j-1) \leftarrow V(j)$

$V(j) \leftarrow temp$

Fim Se

Fim Para

Fim Para

Retornar V

2 Estruturas

Estruturas são tipos de variáveis que agrupam dados geralmente desiguais, enquanto matrizes são tipos de variáveis que agrupam dados similares. O formato da estrutura é definido pelo programador.

Estruturas são usadas normalmente para definir registros a serem armazenados em arquivos.

Os itens de dados de uma estrutura são chamados *membros*. Aprender a lidar com estruturas é o caminho para entender classes e objetos.

2.1 Criando estruturas com *struct*

A definição da estrutura informa como ela é organizada e quais são os seus membros.

```
struct nome_da_estrutura
{
    tipo membro_1;
    tipo membro_2;
    tipo membro_n;
};
```

O identificador *nome_da_estrutura* é o *tag* (marca ou rótulo) da estrutura. Os membros de uma estrutura devem ter nomes diferentes, mas duas estruturas podem ter membros com mesmo nome, sem que haja conflito. Um exemplo de definição de estrutura é visto abaixo.

```
struct aluno{
int numero_mat;
float nota[3];
float media;
};
```

Definir uma estrutura não cria nenhuma variável. Não há reserva de memória. No entanto, a definição da estrutura cria um novo tipo de dado que pode ser usado para criar variáveis.

2.2 Declarando uma variável

A declaração da variável é que reserva espaço de memória suficiente para armazenar todos os membros da estrutura. Ela é realizada segundo a notação já conhecida:

Nome_do_tipo *nome_da_variável*

Portanto, um exemplo de declaração de variável utilizando a estrutura definida na seção anterior é:

struct aluno ANA;

A definição e a declaração da variável podem ser combinadas, colocando-se o nome da(s) variável(eis) após o fechamento da chave e antes do ponto-e-vírgula, como mostrado abaixo.

```
struct aluno{
int numero_mat;
float nota[3];
float media;
}ANA, JOAO;
```

Os membros de uma estrutura podem ser acessados por meio de dois operadores: um operador ponto (·) e o operador de ponteiro de estrutura (→). Nesta apostila trabalharemos somente o operador ponto.

Um membro da estrutura é acessado por meio do nome da variável da estrutura. Por exemplo, para acessar o número da matrícula da aluna ANA, definida anteriormente, devemos escrever:

```
ANA.numero_mat = 456;
```

O programa a seguir mostra um exemplo de uso de estruturas.

```
main()
{
  struct irpf{
  char nome[40];
  char cpf[12];
  float salario;
  };

  struct irpf contribuinte;
  float desc;

  printf("Calculo do IRPF\n\n");
  printf("Informe o nome.....:");gets(contribuinte.nome);
  printf("Informe o cpf.....:");gets(contribuinte.cpf);
  printf("Informe o salario.....:");scanf("%f",&contribuinte.salario);
  printf("\n\n");
  printf("Sr. %s\n", contribuinte.nome);
  printf("CPF no. %s\n", contribuinte.cpf);
  printf("Salario R$ %6.2f\n", contribuinte.salario);
  desc = 27.5*contribuinte.salario/100;
  printf("Desconto IRPF: %5.2f\n", desc);
}
```

2.3 Outras operações com estruturas

- Atribuição só é permitida entre estruturas do mesmo tipo.
- Algumas operações só podem ser realizadas membro a membro.
- Inicializar estruturas é semelhante à inicialização de matrizes.
- Matrizes de estruturas : O processo de declaração de uma matriz de estruturas é análogo à declaração de qualquer outro tipo de matriz.

2.4 Typedef

C permite que o usuário defina novos nomes para os tipos de dados conhecidos, através da palavra-chave typedef. A sintaxe do comando typedef é:

```
typedef tipo novo_nome;
```

onde *tipo* corresponde a qualquer tipo de dado, incluindo o tipo estruturas. Veja os exemplos a seguir.

```
typedef float Numreal;
```

O exemplo acima diz para o compilador reconhecer *Numreal* como outro nome para **float**. Uma boa aplicação do **typedef** é na simplificação da definição de estruturas. Para o programa do cálculo do IRPF, poderíamos usar o comando abaixo logo após a definição da estrutura.

```
typedef struct irpf IRPF;
```

Desse modo, a declaração da variável contribuinte ficaria da seguinte maneira:

```
IRPF contribuinte;
```

A vantagem é que você cria um novo identificador de um tipo composto com um único nome. Certamente essa conveniência valerá a pena quando for preciso declarar uma quantidade razoável de variáveis estrutura, tornando o código mais fácil de ler.

2.5 Enumerações

Enumerações é um conjunto de constantes inteiras que especifica todos os valores possíveis que uma variável desse tipo pode ter. A forma geral de uma enumeração em C é:

```
enum identificador {lista-de-enumeração} lista-de-variáveis;
```

Veja o exemplo da utilização de enum para enumerar os meses do ano.

```
enum mes {jan, fev, mar, abr, mai, jun, jul, ago, set, out, nov, dez};
```

O compilador associa o valor 0 ao mês de janeiro, 1 a fevereiro e assim por diante. O programa a seguir mostra a utilização da declaração acima.

```
enum mes {jan=1, fev, mar, abr, mai, jun, jul, ago, set, out, nov, dez};
main()
{
enum mes MES1,MES2;
MES1 = jan;
MES2 = jul;
if(MES1==MES2)
{
printf("O mes eh o mesmo\n");
printf("%d %d", MES1, MES2);
}
else
{
printf("Sao meses diferentes\n");
printf("%d %d", MES1, MES2);
}
}
```

2.6 Exemplos & exercícios

1. Utilize o tipo de dado estrutura para a criação e preenchimento de uma ficha cadastral para a seleção de alunos a uma bolsa de IC no CESET. Ao final do cadastro, imprima os dados do aluno (a definir) que tem maior CR.
2. Considerando um cadastro de uma locadora de fitas de vídeo (código da fita, nome e gênero do filme), defina uma estrutura apropriada e construa um programa que através de um menu de seleção esteja capacitado a efetuar:
 - (a) o cadastramento das informações e sua classificação pelo código da fita;
 - (b) a pesquisa da fita através do nome;
 - (c) relatório de filmes de acordo com um gênero escolhido.

3 Funções

3.1 Introdução

Função é um conjunto de instruções elaboradas para cumprir uma tarefa particular, tendo um nome para referenciá-las. Utiliza-se funções para dividir uma tarefa original em pequenas tarefas que simplificam e organizam o programa. As principais características das funções são:

- Reduz o tamanho do programa.
- O código de uma função é agregado ao programa uma única vez.
- O controle do programa é desviado para a função e depois volta à instrução seguinte à da chamada da função.
- A estrutura de uma função é semelhante à da função `main()`, que possui este nome especial, sendo a primeira função que o compilador executa.

Pode-se escrever quantas funções forem necessárias dentro de um programa e qualquer função pode chamar qualquer outra.

3.2 Protótipo e definição de funções

O protótipo é a declaração de uma função. Estabelece o tipo de uma função e os argumentos que ela recebe. O protótipo deve ser colocado antes da função `main()`. A sintaxe é a seguinte:

tipo *nome_da_função* (*lista de argumentos*);

O protótipo de uma função deve preceder a sua definição e chamada. O tipo de uma função é determinado pelo valor que ela retorna via comando `return`, e não pelo tipo que ela recebe. Uma função sem `return` é do tipo `void`. A definição é feita semelhante ao protótipo, como mostrado abaixo.

```
tipo nome_da_função (lista de argumentos)
{
corpo da função
}
```

O protótipo nem sempre é necessário. Funções definidas antes de serem chamadas não necessitam de protótipo.

3.3 Passagem de parâmetros

- **Passagem de parâmetros por valor**

De forma geral, os parâmetros (argumentos) são passados para funções por valor, isto é, uma cópia do valor do argumento é passado para a função. Portanto, a alteração do valor recebido pela função não modifica o valor original dos argumentos. Veja os exemplos a seguir, que apresentam uma função do tipo `void` (não retorna nada) e uma função do tipo `int`, pois retorna um valor inteiro associado ao nome da função.

- **Passagem de parâmetros por referência**

Na chamada por referência, o endereço de um argumento é copiado no parâmetro. Então, o endereço é usado para acessar o argumento real utilizado na chamada. Isso significa que alterações feitas no parâmetro afetam variável usada para chamar a rotina. Passagem de parâmetros por referência também possibilita que uma função retorne mais de um valor.

```

// Exemplo - funcao do tipo void - passagem de parametros por valor
// funcao dobro - opcao 1
#include <stdio.h>
void dobro(int valor); // prototipo
main()
{
  int num = 3;
  printf ("%d \n", num);
  dobro(num);
  printf ("%d \n", num);
}

void dobro(int valor)
{
  valor = valor × 2;
  printf ("%d \n", valor);
}

```

```

// Exemplo - funcao do tipo float - passagem de parametros por valor
// funcao dobro - opcao 2
#include <stdio.h>
float dobro2(float valor); // prototipo
main()
{
  float resultado, num = 7.5;
  printf ("%5.2f \n", num);
  resultado = dobro2(num);
  printf ("O dobro de %5.2f eh igual a %5.2f \n", num, resultado);
}

float dobro2(float valor)
{
  return valor × 2;
}

```

```

// Exemplo - funcao do tipo inteiro - passagem de parametros por valor
// calculo do fatorial
#include <stdio.h>
int fatorial(int x); // prototipo
main()
{
  int num, resultado;
  printf ("Digite um número \n");
  scanf ("%d \n", &num);
  resultado =fatorial(num);
  printf ("O fatorial de %d eh igual a %d \n", num, resultado);
}

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

```

Esse tipo de chamada é a adotada principalmente no tratamento de matrizes. Quando uma matriz é usada como argumento para uma função, apenas o endereço da matriz é passado e não uma cópia da matriz inteira. Isso é implementado através do uso de um ponteiro para o primeiro elemento da matriz (o nome da matriz sem qualquer índice, antecedido pelo símbolo *).

3.4 Funções recursivas

Uma função pode chamar a si própria com a finalidade de resolver determinado problema. Essa função é dita *recursiva*. Como a recursão define algo em termos de si mesma, ela também é chamada de *definição circular*.

Na realidade, a função recursiva só sabe resolver o caso mais simples (básico). Quando a função é chamada para uma instância mais complexa, ela divide o problema em duas partes: uma parte que ela sabe resolver e outra que ela não sabe. No entanto, a segunda parte é semelhante ao problema original, mas numa versão pouco mais simples (ou menor) do que ele. Para esse subproblema, a função cria uma nova cópia de si mesma para lidar com o problema menor, conhecido como etapa de recursão. A etapa de recursão inclui **returns** que vão combinando resultados obtidos nas diferentes chamadas à função.

Para a recursão chegar ao fim, os subproblemas devem ser cada vez menores convergindo para um caso básico. Como cada chamada recursiva cria uma cópia da função (das variáveis), isso pode consumir memória considerável.

3.5 Exercícios

1. Desenvolva um programa que crie uma função para calcular o valor de uma potência dados uma base e um expoente quaisquer.
2. Escreva uma função que receba um inteiro positivo e, se este número for primo, retorne 1, caso contrário, retorne 0.

```

// Exemplo - passagem de parametros por referencia
// transformacao de string minusculo para maiusculo
#include <stdio.h>
#include <string.h>
#include <conio.h>
void strmaius(char *s); // prototipo
//void strmaius(char s[]); tambem poderia ter sido usado esse prototipo
main()
{
char s1[20];
printf ("\n Digite uma string (maximo de 20 caracteres): ");
gets (s1);
strmaius (s1);
getch();
}

void strmaius(char *s)
//void strmaius(char s[]) tambem poderia ter sido usada essa definicao
{
int tamanho =strlen(s);
printf ("\n\t Em maiusculo...");
for(int i = 0;i < tamanho;i++)
printf("%c",s[i]>96 && s[i]<123) ? char (s[i]-32) : s[i]);
}

```

```

// Exemplo - passagem de parametros por referencia
// calculo do dobro de um numero
#include <stdio.h>
#include <conio.h>
void dobro(float *num); // prototipo
main()
{
float numero;
printf ("\n Digite um numero: ");
scanf ("%f",&numero);
printf ("\n O dobro de %5.2f ", numero);
dobro (&numero);
printf ("\n eh igual a %5.2f \n", numero);
getch();
}

void dobro(float *num)
{
*num=*num*2;
}

```

```

// Exemplo - função recursiva
// cálculo do fatorial
#include <stdio.h>
#include <stdlib.h>
int fat(int n); // prototipo
int main()
{
    int j;
    printf ("Digite um número: \n");
    scanf ("%d \n", &j);
    printf ("O fatorial de %d eh igual a %d \n", j, fat(j));
    return 0;
}

int fat(int n)
{
    if(n == 1 || n == 0)
        return 1;
    else
        return fat(n - 1) * n; //chamada recursiva
}

```

3. Suponha que você desenvolveu um editor de textos. Desenvolva um programa em forma de funções que tenha as seguintes opções:
 ESTATÍSTICAS
 (a) CONTAR CARACTERES
 (b) CONTAR PALAVRAS
 (c) FINALIZAR
4. Suponha que os preços de um mercado serão reajustados em 20%. Elabore um programa com função para calcular o valor reajustado de um produto e o valor do reajuste, apresentando-os em seguida.
5. Encontrar o maior elemento de uma lista $L[1..n]$. São apresentados dois algoritmos: o primeiro (Algoritmo 2) é considerado como sendo uma solução ingênua, simples; e o segundo (Algoritmo 3), trata o problema através da método recursivo. No segundo algoritmo, x e y correspondem aos índices inicial e final da lista L . Implemente as duas soluções em C.

Algoritmo 2 Determina o maior elemento em uma lista - Solução ingênua

```
max ← L(1)
Para i de 2 até n Faça
  Se L(i) > max Então
    max ← L(i)
  Fim Se
Fim Para
```

Algoritmo 3 Determina o maior elemento em uma lista - Solução por divisão e conquista

```
function Maximo(L(x, y))
Se  $y - x \leq 1$  Então
  return max(L(x), L(y))
Se Não
  max1 ← Maximo(L(x, (x + y) / 2))
  max2 ← Maximo(L((x + y) / 2 + 1, y))
  return max(max1, max2)
Fim Se
```

4 Arquivos

4.1 Introdução

Arquivos são usados para conservação permanente de grande quantidade de dados. Um arquivo é um conjunto de registros (**struct**, em C), que por sua vez é um conjunto de campos. Um campo é um conjunto de caracteres que possuem um significado, ou seja, é a variável isolada com a qual trabalhamos desde o início deste Curso, como por exemplo, a variável que recebe o nome de uma pessoa, uma profissão, um salário etc.

Há duas maneiras de acesso aos dados presentes em um arquivo: seqüencial e aleatório. A forma mais comum é o acesso seqüencial, na qual os registros são armazenados numa ordem segundo um campo chave. Por exemplo, no arquivo de uma lista de alunos de uma determinada disciplina, o campo chave poderia ser o nome do aluno ou o seu RA.

Um grupo de arquivos relacionados entre si é chamado de *banco de dados*. Um conjunto de programas que se destina a criar e gerenciar banco de dados é chamado de *Sistema de Gerenciamento de Banco de Dados (SGBD)*.

A linguagem C visualiza um arquivo como um fluxo seqüencial de bytes. Cada arquivo tem um marcador de final de arquivo (*end-of-file marker*).

Abrir um arquivo retorna um ponteiro para uma estrutura **FILE** (definida em **stdio.h**) que contém as informações usadas para processar o arquivo.

4.2 Arquivo de acesso seqüencial

4.2.1 Criando arquivos

Para se manipular arquivos é necessário se efetuar duas operações básicas: abertura (**fopen()**) e fechamento(**fclose()**). A Tabela 2 mostra o tipo de abertura de arquivos em C.

Tipo	Descrição
r	abre arquivo para leitura.
w	abre arquivo para escrita (gravação). Caso o arquivo exista, o arquivo será recriado, apagando o anterior.
a	abre arquivo para escrita, acrescentando dados ao final do mesmo. Caso o arquivo não exista, ele será criado.

Tabela 2: Tipos de abertura de arquivos.

O programa a seguir mostra um exemplo simples de criação de um arquivo denominado ARQEXE01.DAT.

```
// Exemplo - criacao de um arquivo
#include <stdio.h>
main()
{
    FILE *ptrARQ; //cria um ponteiro para um arquivo
    ptrARQ = fopen("C:/TEMP/ARQEXE01.DAT", "a");
    //a linha acima cria o arquivo ARQEXE01.DAT no diretorio TEMP
    fclose(ptrARQ); //fecha o arquivo criado
}
```

4.2.2 Gravando dados em um arquivo

Após a criação de um arquivo, este pode ser aberto para que informações possam ser armazenadas. O programa a seguir ilustra a gravação de uma string no arquivo ARQEXE01.DAT.

```
// Exemplo - gravacao em arquivo
#include <stdio.h>
main()
{
  char nome[20];
  FILE *ptrARQ; //cria um ponteiro para um arquivo
  ptrARQ = fopen("C:/TEMP/ARQEXE01.DAT","w"); //abre arquivo para gravacao
  printf ("Digite um nome: \n");
  scanf ("%s", &nome);
  fprintf (ptrARQ,"%s",nome); //escreve nome no arquivo
  fclose(ptrARQ); //fecha o arquivo criado
}
```

4.2.3 Lendo dados de um arquivo

Completando a seqüência de operações, é apresentado a seguir um programa que efetua a leitura dos dados gravados pelo programa da seção anterior.

```
// Exemplo - leitura de arquivo
#include <stdio.h>
main()
{
  char nome1[20];
  FILE *ptrARQ; //cria um ponteiro para um arquivo
  ptrARQ = fopen("C:/TEMP/ARQEXE01.DAT","r"); //abre arquivo para leitura
  fscanf (ptrARQ,"%s",nome1); //lê nome1 no arquivo
  printf ("Palavra armazenada no arquivo: %s\n",nome1);
  fclose(ptrARQ); //fecha o arquivo criado
}
```

4.3 Exemplos mais elaborados

O programa ABREARQ a seguir ilustra duas novas operações: o teste de abertura de arquivo e a saída do programa através da função `exit()`. Esta função provoca o encerramento imediato do programa, forçando um retorno ao sistema operacional. O código de retorno, colocado entre parênteses, indica normalmente algum tipo de erro. O zero é geralmente usado como um código de retorno que indica uma terminação normal do programa.

Já o programa LETRA abaixo exemplifica o uso de argumentos de entrada na função `main()`. Além disso, utiliza-se outras funções de leitura e escrita de dados no arquivo. O programa LETRA lê caracteres via teclado e os escreve no arquivo criado até ser digitado o símbolo de cifrão \$. O nome do arquivo a ser criado deve ser especificado na linha de comando. Por exemplo, deve-se digitar LETRA ARQLETRA.DAT para escrever os caracteres no arquivo ARQLETRA.DAT.

Para executar o programa LETRA, abra uma janela no DOS, vá para a pasta que contém o arquivo executável e digite, por exemplo, LETRA teste.dat.

```

// Exemplo - programa ABREARQ
// Exemplo - teste abertura e funcao exit()
#include <stdio.h>
main()
{
int RA;
char nome[20];
float media;
FILE *ptrARQ; //cria um ponteiro para um arquivo
// Testa se eh possivel abrir o arquivo
if((ptrARQ = fopen("c:/temp/ARQEXE04.DAT", "w"))==NULL)
{
printf("Arquivo nao pode ser aberto \n");
exit(0); //retorna ao sistema operacional
}
// le dados
printf ("\n Digite nome aluno: "); scanf ("%s", nome);
printf ("\n Digite RA aluno: "); scanf ("%d", &RA);
printf ("\n Digite media aluno: "); scanf ("%f", &media);
fprintf (ptrARQ, "%s %d %f", nome, RA, media); //escreve dados no arquivo
fclose(ptrARQ); //fecha o arquivo criado
}

```

```

// Exemplo - programa LETRA
// Exemplo - uso de argumentos em main() e nova funcao de escrita de dados
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[])
{
char caracter;
FILE *ptrARQ; //cria um ponteiro para um arquivo
// Testa se foi digitado um nome para o arquivo de saida
if(argc != 2)
{
printf("Vc nao digitou o nome do arquivo \n");
exit(1); //termina a execucao do programa
}
// Testa se eh possivel abrir o arquivo
if((ptrARQ = fopen(argv[1], "w"))==NULL)
{
printf("Arquivo nao pode ser aberto \n");
exit(0); //retorna ao sistema operacional
}
printf ("\n Digite caracteres...($ para sair: ");
do {
caracter =getchar(); // le dados
putc (caracter, ptrARQ); //escreve dados no arquivo
}while(caracter != '$');
fclose(ptrARQ); //fecha o arquivo criado
}

```

argc e **argv** são dois argumentos internos especiais, usados para receber os argumentos da linha de comando. **argc** contém o número de argumentos (número inteiro), sendo que o nome do programa também é qualificado como primeiro argumento. **argv** é um ponteiro que aponta para cada argumento da linha de comando. No programa exemplo, **argv[0]** aponta para o nome do programa e **argv[1]**, para o nome do arquivo.

4.4 Exercícios

1. Desenvolva um programa em C que escreva os números inteiros de 1 a 100 em um arquivo de nome INTEIRO.DAT.
2. Dada uma lista de notas da disciplina ST263 em um arquivo de nome ST263.DAT, construa um programa que leia as notas e calcule a média da turma. O número de alunos não é conhecido. O formato das notas no arquivo para uma turma de 10 alunos é ilustrado abaixo.

10
8.3
0.5
6.8
7.2
4.9
9.1
3.3
8.0
7.4

5 Alocação dinâmica

5.1 Introdução

Alocar memória dinamicamente significa obter durante a execução do programa mais espaço em memória para o armazenamento de novas variáveis. Alocação dinâmica também implica em liberar memória que não está sendo mais utilizada pelo programa.

Alocação dinâmica é útil quando não se conhece o tamanho do vetor que será trabalhado, em tempo de compilação, ou quando se quer criar e destruir variáveis em qualquer parte do programa.

As funções relacionadas à alocação dinâmica estão na biblioteca **stdlib.h**: **malloc** e **free**. Além disso, utiliza-se também o operador **sizeof**. A função **malloc** aloca memória (tamanho em bytes, calculado normalmente pelo operador **sizeof**) e retorna um ponteiro para a memória alocada. A função **free** libera a memória alocada, deixando-a disponível ao sistema para que possa ser reutilizada.

5.2 Alocando memória dinamicamente para um vetor

O programa a seguir aloca memória dinamicamente para um vetor de tamanho a ser escolhido pelo usuário. Em seguida, ele soma os elementos do vetor e apresenta o resultado.

```
// Exemplo - programa ALOCACAO
// Exemplo - alocacao dinamica em C
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *vetor, num, soma=0;
    printf("Digite numero de elementos do vetor:");
    scanf("%d",&num);
    //tenta alocar memoria - retorna um ponteiro do tipo inteiro
    vetor=(int *)malloc(num*sizeof(int));
    if(vetor==NULL) //testa se ha memoria disponivel para alocacao
    {
        printf("Nao ha memoria disponivel!");
        return 1; // sai do programa
    }
    printf("Digite um vetor (%d elementos): \n",num);
    for(int i=0;i<num;i++)
    {
        printf("V[%d]= ",i+1);
        scanf("%f",&vetor[i]);
        soma += vetor[i];
    }
    printf("Soma dos elementos do vetor = %d ", soma);
    free(vetor); //libera memoria alocada
    return 0;
}
```

A alocação de memória para o vetor é efetuada, se possível, na linha do **malloc**. Caso se digite *num*=4, por exemplo, será liberado 4 vezes o espaço de memória para armazenar uma variável inteira (2 *bytes*), ou seja, um total de 8 *bytes*. No programa, **NULL** é o valor inicial de um ponteiro que não aponta para lugar algum. Ele corresponde a uma constante simbólica, definida em **stdlib.h**, e é conhecido como *ponteiro nulo*.

Referências

- [1] Schildt, H. C Completo e Total. São Paulo: Makron Books, 1996.
- [2] Deitel, H. M., Deitel, P. J. Como Programar em C. Rio de Janeiro: LTC, 1999.
- [3] Arnush, C. Teach Yourself Turbo C++ 4.5 For Windows in 21 Days. Indianapolis: Sams Publishing, 1998.
- [4] Manzano, J. A. N. G. Linguagem C - Estudo Dirigido. São Paulo: Érica, 1997.
- [5] CÔRTEZ, P. L. Turbo C: Ferramentas & Utilitários, vol. 1. São Paulo: Érica, 1999.
- [6] Lehmann, A.H., Trevisan, R. Turbo C. Guia de Referência Rápida de todas as Funções. São Paulo: Érica, 1990.