É mais difícil comandar do que obedecer. F. Nietzsche.

PASSAGEM DE PARÂMETROS

Como visto no laboratório 9, uma função é composta por quatro partes distintas:

- Nome da função
- Tipo de retorno
- Lista de parâmetros
- Corpo da função

PT1: O programa a seguir cria uma função Maior cuja lista de parâmetros contém dois números inteiros a e b e que retorna o resultado de max{a,b}.

```
#include <stdlib.h>
#include <stdio.h>
int Maior(int a, int b)
{
  return (a > b) ? a:b;
}

main()
{
   int a, b, max;
   printf("Entre com a e b:");
   scanf("%d %d",&a,&b);
   max = Maior(a,b);
   printf("Max{a,b} = %d \n",max);
}
```

A função Maior do **PT1** possui as seguintes propriedades:

Nome da função	Maior
Tipo de retorno	int
Lista de parâmetros	int a, int b
Corpo da função	**return** (a>b)? a:b;

Observe que o número e o tipo dos argumentos passados à função devem corresponder ao número e ao tipo dos parâmetros com que esta foi definida.

Uma observação importante é que uma função que não retorne qualquer tipo, isto é que retorne **void**, denomina-se de procedimento.

PE1: Modifique a função **Maior** do **PT1** para que ao invés de retornar o maior entre dois valores, apenas imprima na tela o maior valor, e portanto tenha tipo de retorno igual a **void**.

PASSAGEM POR VALOR

A troca de valores entre variáveis é um problema freqüente de programação e pode ser usado na ordenação de um vetor. Uma troca é realizada em três passos e sempre com o auxílio de uma variável auxiliar.

PT2: O programa a seguir cria uma função troca que realiza a troca de valores entre duas variáveis.

```
#include <stdio.h>
void troca(int a, int b); // protótipo
main()
{ int a, b;
  puts("Dois nums inteiros: ");
  scanf("%d %d", &a, &b);
  printf("Antes de troca \n");
  printf("a= %d e b=%d \n",a,b);
  troca(a,b);
  printf("Depois de troca \n");
  printf("a= %d e b=%d \n",a,b);
}
void troca(int a, int b)
{
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
}
```

Observe que no **PT2** não é realizada a troca de valores. Para entender o que aconteceu, é necessário compreender dois conceitos:

- Escopo de variáveis.
- Passagem por valor e referência.

Quando a função **main()** é executada, são criadas as variáveis a e b. Por terem sido declaradas na função **main()** sua visibilidade (ou seja, o escopo para se manipular estas) é restrita à esta função. Quando a função **troca()** é invocada, novas variáveis a e b, visíveis apenas dentro da função **troca()**, são criadas, suspendendo temporariamente a visibilidade de a e b criadas em **main()**. A chamada de **troca()** em **main()** porém, resulta na **passagem** da cópia **dos valores** de a e b de **main()** para as novas variáveis a e b que são criadas em troca(). Esta situação corresponde ao seguinte esquema:

Função main()	Função troca()
a = 1;	a = 1;
b = 4;	b= 4;
troca(a,b);	comandos;
a=1;	a = 4;
b=4;	b = 1;

Assim, dentro da função **troca()** quaisquer alterações de a e b não resultarão em modificações de a e b em **main()**, pois o que está sendo alterado é apenas uma cópia.
Ou seja, o **PT2** corresponde a um programa onde é realizada a passagem de parâmetros por valor, sendo enviadas para a função cópias dos valores originais.

PASSAGEM POR REFERÊNCIA

Uma forma alternativa é enviar parâmetros por referência, ou seja, o que é enviado para a função não é uma cópia do valor da variável e sim uma referência a esta. Assim, qualquer alteração dos parâmetros realizada na função corresponde a uma alteração nas variáveis originais. Para realizar a passagem por referência ponteiros devem ser utilizados. A estratégia a ser adotada é:

(1)Passar o endereço de uma variável, obtido através do operador & e armazenado em uma variável do tipo ponteiro.

(2) Dentro da função, usar ponteiros para alterar os locais para onde eles apontam e que correspondem aos valores das variáveis originais.

PT3: O programa a seguir reescreve o **PT2** utilizando a passagem de parâmetros por referência.

```
#include <stdio.h>
void troca(int *a, int *b); // protótipo
main()
{ int a, b;
  puts("Dois nums inteiros: ");
  scanf("%d %d", &a, &b);
  printf("Antes de troca \n");
  printf("a= %d e b=%d \n",a,b);
  troca(&a,&b);
  printf("Depois de troca \n");
  printf("a= %d e b=%d \n",a,b);
}
void troca(int *a, int *b)
{
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Note que a variável **tmp** continua a ser declarada do tipo **int**, pois seu propósito é armazenar um dos valores e não endereço.

PASSAGEM DE VETORES

Diferentemente das variáveis em que existe a passagem por valor ou referência, a passagem de vetores para funções é sempre por referência. A sintaxe de passagem de um vetor pode ser feita com:

```
tipo nome[ ];
```

onde: tipo – corresponde ao tipo dos elementos do vetor, nome – é o nome atribuído ao vetor e [ ] indica que a variável é do tipo vetor. O "[ ]" pode ser utilizado sem um valor, pois em C não interessa qual a dimensão do vetor que é passado a uma função, mas sim o tipo dos seus elementos.

PT4: Este programa cria uma função **inic()** que inicia um vetor de inteiros e teste a mesma.

```
#include <stdio.h>
void inic(int s[ ], int n)
{ int i;
  for (i=0;i<n;i++)
    s[i]=0;
}
imprimeV(int t[ ], int n)
{ int i;
  for(i=0; i < n; i++)
     printf(" [%d] ",t[i]);
  printf("\n");
}

main()
{ int v[10], i;

  inic(v,10);

  // Impressao apos inicializacao.
  imprimeV(v,10);

  // Mudando valores de v.
  for(i=0;i<10;i++)
    v[i]=i;

  // Impressao apos atribuicao.
  printV(v,10);
}
```

Note que a função **void** inic(**int** s[ ], **int** n) recebe um vetor de inteiros (sem indicar qual a sua dimensão) e um inteiro que indica qual o número de elementos a iniciar. **Compile o programa PT4 e verifique que ele contém um erro de link. Descubra porque ocorre esse erro e arrume o programa para que funcione corretamente**

PE2: Implemente e teste a função **void** max(**float** v[ ], **int** n, **float** *max) que recebe um vetor de números reais e o número de elementos a considerar e calcule o maior número dentre os n primeiros elementos do vetor, mandando esse valor para o main usando passagem de parâmetros por referência (ponteiro).

A passagem de vetores com mais de uma dimensão para uma função é realizada indicando no cabeçalho desta, obrigatoriamente, o número de elementos de cada um das n-1 dimensões à direita. Apenas a dimensão mais à esquerda pode ser omitida, colocando-se [] ou um asterisco. É, no entanto, habitual colocar todas as dimensões dos vetores.

PT5: Construir uma função initM() que inicializa uma matriz e outra função mostraM() que mostra os elementos de uma matriz.

```
#include <stdio.h>
#define DIM 3
void initM( int s[ ] [DIM] ) // sem 1 DIM
{
  int i, j;
  for (i=0;i<DIM;i++)
    for (j=0; j <DIM;j++)
      s[i][j] = 0;
}
void mostraM( int s[DIM] [DIM])//2DIM
{
  int i, j;
  for (i=0; i<DIM;i++)
  {
    for (j=0;j<DIM;j++)
      printf(" [%d] ",s[ i ][ j ]);
    printf("\n");
  }
}
main()
{
  int s[DIM][DIM];
  initM(s);
  mostraM(s);
}
```

PE3: Adicione uma nova função modEle ao **PT5** cujos parâmetros são s, i, j e num, tal que modifica o elemento s[i][j] com o valor num. Teste para i =1, j = 2 e num = 10.

PE4: Modifique o **PT5** para que a função mostraM ao invés de mostrar os valores inteiros contidos em s, mostre os caracteres associados a cada valor inteiro positivo. **Dica:** use uma conversão explícita com (**char**)s[i][j] e mude a função initM para inicializar os elementos de s com o valor 1.

Uma outra forma de manipular os valores contidos em um vetor através de funções é utilizando ponteiros. Para tanto, basta lembrar que se v for um vetor ou um ponteiro para o primeiro elemento de um vetor (lembre-se que p = v ↔ p = &v[0]), então o elemento cujo índice é i pode ser obtido usando v[i] ou *(v+i) . Ou seja:

v[i] ↔ *(v+i)

Isto só é possível, pois quando um vetor é declarado, os seus elementos são alocados em espaços vizinhos de memória:

EXEMPLO **E1**:
char v[10] = "Ola";

v	v+1	v+2	v+3
'O'	'l'	'a'	'\0'
1000	1001	1002	1003

É importante ter em mente que sempre que uma função é invocada utilizando um vetor como um parâmetro, o vetor não é recebido em sua totalidade, mas apenas o endereço inicial do vetor, afinal v ↔ &v[0].
Por esse motivo, o vetor que é definido no cabeçalho de uma função pode ser referenciado através de um ponteiro:

PT6: Construir e testar uma função que fornece o tamanho de uma String.

```
#include <string.h>
#include <stdio.h>
int strlen(char *s)
{ char *ptr = s; // guardar  endereço inicial
  while (*s != '\0') // contar até o fim
    s++;
  return (int) (s-ptr);
}
main()
{  char Nome[100];
   printf("Digite uma String: "); gets(Nome);
   printf("length = %d \n",strlen(Nome));
}
```

Utilizando o Exemplo **E1**, observe que ao final do **while** da função **strlen** o valor de s será 1003 ao passo que **ptr** terá o valor 1000. Ao se fazer (s-ptr) o resultado é 3, o que corresponde ao tamanho da String (lembre que 1 caractere ocupa 1 byte na memória).

CRIAÇÃO DE ESTRUTURAS

Uma estrutura é um conjunto de uma ou mais variáveis (também chamadas campos ou membros), agrupadas sob um único nome, de forma a facilitar a sua referência. As estruturas podem conter elementos de qualquer tipo de dados válidos em C (tipos básicos: **char**, **int**, **float**, **double**, **strings**, vetores, ponteiros ou mesmo estruturas). Sua sintaxe é dada por:

```
struct  nome_da_estrutura
{
  tipo1 campo1, campo2;
   ...
  tipon campo;
};
```

A declaração de uma estrutura corresponde unicamente à definição de um novo tipo (isto é, da sua estrutura), e não à declaração de variáveis do tipo estrutura. Um exemplo de declaração de uma estrutura capaz de armazenar datas é dado por:

```
struct Data
{
 int Dia, Mes, Ano;
};
```

A definição de **Data** (**struct Data**) indica simplesmente que, a partir daquele momento o compilador passa a conhecer um outro tipo, chamado **struct** Data, que é composto por tres inteiros. Ou seja, **struct Data não é uma variável e sim o nome pelo qual é conhecido o novo tipo de dados**.

DECLARAÇÃO DE VARIÁVEIS DO TIPO ESTRUTURA

Para declarar uma variável do tipo **struct Data**, apresentada anteriormente, basta indicar qual o tipo (**struct Data**) seguido do nome das variáveis:

```
struct Data d;
```

Ou seja, a variável d é do tipo **struct Data**. Alternativamente, a definição de uma estrutura e a declaração de variáveis poderia ser feita simultaneamente de acordo com a seguinte sintaxe:

```
struct nome_da_estrutura
{
  tipo1 campo1, campo2;
   ...
  tipon campo;
} v1, v2, ..., vm;
```

Para criar variáveis d1, d2 do tipo Data:

EX1: Criar variáveis d1 e d2 do tipo Data:

```
struct Data {int Dia, Mes, Ano;}d1, d2;
```

EX2: Crie uma estrutura que define um indivíduo. Declare as variáveis P1 e P2 na definição da Estrutura.

```
struct Pessoa
{int idade;
 char sexo, est_civil;
 char Nome[60];
 float salário;
} P1, P2;
```

Observe que uma estrutura pode agregar diferentes tipos básicos (**int**, **float**, **double**, **char**). Além disso, ao se definir variáveis do tipo **struct** Pessoa os conteúdos dos campos idade, sexo, est_civil e Nome são diferentes para cada variável. Ou seja, o campo nome da variável P1 pode conter a string "Paulo Silva" e o campo nome da variável P2 pode conter a string "Teresa Maria".

CARGA INICIAL AUTOMÁTICA DE ESTRUTURAS

Uma estrutura pode ser iniciada quando é declarada usando a sintaxe:

```
struct nome_estrutura var = {valor1,...,valorn};
```

Entre chaves são colocados os valores dos membros da estrutura pela ordem em que esses foram escritos na definição.

EX3: Declare e inicialize uma variável dt do tipo declarado pela estrutura Data com os campos dia, mes e ano inteiros definidos nesta ordem:

```
// Declaração da estrutura Data e da
// variável dt do tipo struct Data.
struct Data {int dia, mes, ano; }dt = {23, 1, 1966};
// Apenas Declaração da estrutura Data.
struct Data {int dia, mes, ano; };
// Declaração e inicialização da variável dt.
struct Data dt = {23,1,1966};
```

A descrição do acesso e modificação dos membros de uma estrutura é dada a seguir.

MANIPULAÇÃO DE VARIÁVEIS DO TIPO ESTRUTURA

Para acessar o membro mmm de uma estrutura eee usa-se o operador ponto (.) fazendo eee.mmm. Por exemplo:

PT1: Inicialização de uma estrutura Data.

```
// Criação e declaração de Data.
#include <stdio.h>
struct Data{int dia, mes, ano;} dt;

main()
{ // Atribuição de valores a cada membro.
  dt.dia = 23;
  dt.mes = 1;
  dt.ano = 1966;
  // Impressão dos valores.
  printf("Data:  %d/%d/%d\n",  dt.dia, dt.mes, dt.ano);}
```

DEFINIÇÃO DE TIPOS - TYPEDEF

Uma desvantagem na utilização de estruturas está na declaração de variáveis, que têm sempre que ser precedidas da palavra reservada **struct**, seguida do nome da estrutura. Isto pode ser evitado usando-se a palavra reservada **typedef** cuja sintaxe é:

```
typedef  tipo_existente  sinônimo;
```

A palavra **typedef** não cria um novo tipo, e sim permite apenas que um determinado tipo possa ser denominado de forma diferente, com um sinônimo, de acordo com as necessidades do programador. A palavra **typedef** não se limita apenas às estruturas, mas pode ser utilizada com qualquer tipo da linguagem.

PT2: Crie um novo nome para o tipo inteiro (**int**) e realize alguma operação com a variável declarada.

```
typedef int inteiro;
main()
{inteiro a;
  a = 1;
  printf("a = %d",a);}
```

PT3: Declare um novo tipo denominado PESSOA, que permita identificar a estrutura Pessoa e crie variáveis de duas formas.

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

typedef  struct Pessoa
{  int idade;
   char sexo, est_civil;
   char Nome[60];
   float salario;
} PESSOA;

main()
{ struct Pessoa P1= {19, 'F', 'S', "Clara", 2650.00 };
PESSOA P2;
// Entrada dos valores de P2.
printf("Entre com o nome:");
scanf("%s", P2.Nome);
printf("\n Idade: ");
scanf("%d",&P2.idade);
printf("\n Salario: ");
scanf("%f",&P2.salario);
fflush(stdin);
printf("\n Sexo:");
P2.sexo  = getche();
printf("\n Estado Civil:");
P2.est_civil  = getche();
// Saída de valores.
if (P2.idade > P1.idade)
  printf("\n %s eh tiozinho ! ", P2.Nome);
else
  printf("\n %s eh tiozinho ! ", P1.Nome);
}
```

Não é possível ao mesmo tempo definir um **typedef** e declarar variáveis, tal como realizado para a definição de **struct** no **EX1** e **EX2**. Se o sinônimo será sempre utilizado na definição de tipos e variáveis, então, a estrutura pode ser declarada sem o nome:

PT4: Refazer o **PT3** sem declarar um nome.

```
typedef struct
{  int idade;
   char sexo, est_civil;
   char Nome[60];
   float salário;
} PESSOA;
```

Observe a definição de struct do **PT4** com o **PT3** e diga quais são as diferenças entre eles.

ONDE DEFINIR ESTRUTURAS E TYPEDEF

A definição das estruturas deve ser realizada fora de qualquer função (inclusive a função main()) – em geral no início do programa de acordo com a seguinte ordem:

```
/* Inclusão de bibliotecas */
#include <...>
/* Definição de novos tipos*/
struct data {...};
typedef ...;
```

```
/* Protótipos das funções */
f(...);
/* Função Principal */
main(){...}
/* Definição das funções */
f(){...}
```

PT5: Escreva um programa que escreva na tela os valores dos campos que compõem uma estrutura Pessoa (cujo **typedef** é **PESSOA**) e cujos campos são: Nome(**char**), idade(**int**) e salario(**float**). Dica: Use o **PT3**.

```
#include <stdio.h>
typedef struct Pessoa
{
  char nome[100];
  int idade;
  float salario;
} PESSOA;

main()
{
   struct Pessoa p1 = {"Carlos",23,8500};
   PESSOA p2 = {"Maria",21,8510};

   // Mostrando os valores dos campos.
   // Para a pessoa p1.
   printf("Nome: %s\n",p1.nome);
   printf("Idade: %d\n",p1.idade);
   printf("Salario: %.2f\n",p1.salario);
   // Para a pessoa p2.
   printf("Nome: %s\n",p2.nome);
   printf("Idade: %d\n",p2.idade);
   printf("Salario: %.2f\n",p2.salario);
}
```

VETORES E MATRIZES DE ESTRUTURAS

Uma estrutura pode ser aplicada em conjunto com um vetor ou uma matriz de forma a simplificar o armazenamento e o acesso de informação de um novo tipo. Um exemplo é o **PT6** que simplifica o **PT5**.

PT6: Refazer o **PT5** de forma a empregar um vetor cujos elementos são do tipo **struct Pessoa** tal como definido no **PT5**.

```
#include <stdio.h>
typedef struct Pessoa
{
  char nome[100];
  int idade;
  float salario;} PESSOA;

const int n = 2;
main()
{  int i;
   PESSOA Vp[n];

   // Laço para leitura dos dados.
   for (i=0; i < n; i++)
   {  printf("\n Entre com nome: ");
      gets(Vp[i].nome);
      printf("\n Entre com a idade: ");
      scanf("%d",&Vp[i].idade);
      printf("\n Entre com o salario: ");
      scanf("%f",&Vp[i].salario);
      fflush(stdin); }

   // Laço impressão valores dos campos.
   for (i=0; i < n; i++)
   { printf("Nome: %s\n",Vp[i].nome);
     printf("Idade: %d\n",Vp[i].idade);
     printf("Salario: %.2f\n",Vp[i].salario);  }
}
```

O **PT6** introduz uma facilidade em relação ao **PT5** que é capacidade de armazenar e acessar n informações do tipo **struct Pessoa** apenas empregando o índice de um vetor.

FUNÇÕES E ESTRUTURAS

Após a definição de um novo tipo de variável a partir do comando **struct** e **typedef** é possível empregar o novo tipo também para a passagem de valores entre funções, tanto como argumento de entrada como valor retornado pela função. O **PT10** ilustra essa possibilidade.

PT7: Refazer o **PT6** de forma a empregar as funções **void** leitura(**PESSOA** V[ ]) e **void** impressao(**PESSOA** V[ ]) para leitura e impressão dos dados de V do tipo struct, respectivamente.

```
// Bibliotecas.
#include <stdio.h>
// Definição do tipo PESSOA.
typedef struct Pessoa
{
  char nome[100];
  int idade;
  float salario;
} PESSOA;

// Protótipos das funções.
void leitura(PESSOA Vp[]);
void impressao(PESSOA Vp[]);

// Definição do tamanho do vetor V.
const int n = 2;
```

```
// Função principal.
main()
{
   PESSOA V[n];
   leitura(V);
   impressao(V);
}
// Função para leitura dos dados.
void leitura(PESSOA Vp[])
{  int i;
   // Laço para leitura dos dados.
   for (i=0; i < n; i++)
   {  printf("\n Entre com nome: ");
      gets(Vp[i].nome);
      printf("\n Entre com a idade: ");
      scanf("%d",&Vp[i].idade);
      printf("\n Entre com o salario: ");
      scanf("%f",&Vp[i].salario);
      fflush(stdin);}
}

// Função para impressão dos dados.
void impressao(PESSOA Vp[])
{  int i;
   // Laço impressão valores dos campos.
   for (i=0; i < n; i++)
   { printf("Nome: %s\n",Vp[i].nome);
      printf("Idade: %d\n",Vp[i].idade);
      printf("Salario: %.2f\n",Vp[i].salario);
   }
}
```

Observe que qualquer modificação do vetor Vp[] nas funções leitura e impressao modifica também os valores do vetor v contido na função main(). Observe ainda que a palavra **PESSOA** passa a ser tratada pelo compilador como a que identifica um novo tipo de variável. Tal característica pode ser observada nos protótipos e ou cabeçalhos das funções leitura e impressao.

Embora nenhuma das funções empregue este recurso, uma função também poderia retornar o tipo **PESSOA**, bastando realizar duas coisas:
(i) O tipo de retorno deve ser **PESSOA**:

```
// Protótipo.
PESSOA  funcao1(...);
```

(ii) A função deve retornar efetivamente um valor correspondente ao tipo PESSOA:

```
PESSOA funcao1(PESSOA P1)
{
  return P1;
}
```

ARQUIVOS

A motivação para utilizar arquivos é que o armazenamento de dados em variáveis, vetores e estruturas declaradas em um programa é temporário. A conservação permanente de grandes quantidades de dados deve ser realizada em dispositivos secundários tais como discos magnéticos, discos óticos e fitas. Para tanto, é necessário empregar o conceito de arquivo:

• Arquivo físico: seqüência de bytes armazenados em disco. Um disco pode conter centenas de milhares de arquivos físicos distintos.
• Arquivo lógico, **stream** ou descritor: é o arquivo segundo o ponto de vista do aplicativo que o acessa.

Uma **stream** é o fluxo de dados de um programa para um dispositivo como um arquivo físico ou outros dispositivos de E/S: monitor de vídeo e teclado. Neste processo, existe uma área da memória que é usada temporariamente para armazenar os dados, antes de enviar os mesmos para o seu destino, chamada de **buffer**.
Com a ajuda do **buffer**, o sistema operacional pode aumentar a sua eficiência, reduzindo o número de acessos aos dispositivos E/S. Por padrão, todas as **streams** utilizam o **buffer**.

OPERAÇÕES COM ARQUIVOS

Para abrir um arquivo qualquer, a primeira operação a ser realizada é ligar uma variável de um programa (**stream**) a um arquivo (físico). Isto pode ser realizado através da criação de uma variável do tipo ponteiro para o tipo **FILE** (definido no arquivo <stdio.h>) que será associada a um arquivo através do comando **fopen** de acordo com a seguinte sintaxe:

```
FILE *arq;
arq = fopen("nome_arq", "modo" );
```

onde: "nome_arq" é o nome do arquivo que se quer abrir (por exemplo, "arquivo.dat" ou "Z:\\LP\\arquivo.dat") de acordo com "modo". Existem os seguintes modos de abertura, descritos na **Tabela T1**:

Modo	Descrição
"r"	Leitura, caso não possa abrir devolve NULL.
"w"	Escrita, cria ou sobrescreve um arquivo já existente. Caso não possa criar devolve NULL.
"a"	Acrescenta, coloca no final do arquivo os novos dados. Caso o arquivo já exista, funciona como "w".
"r+"	Abertura para leitura e escrita.
"rt"	Leitura de arquivo texto.
"rb"	Leitura de arquivo binário.

Tabela T1: Modos de leitura.

O comando para se fechar um arquivo é **fclose**. Ao se fechar um arquivo a ligação entre a variável e o arquivo existente no disco é encerrada e todos os dados que possam existir em **buffers** associados ao arquivo são gravados fisicamente em disco (**flushed**). A memória alocada pela função **fopen** para a estrutura do tipo **FILE** é liberada. A sintaxe da função **fclose** é:

```
fclose(FILE *arq);
```

PT1: Construa um programa que indique se um determinado arquivo pode ou não ser aberto.

```
#include <stdio.h>
main()
{
FILE *fp;
char s[100];
puts("Introduza o Nome do Arquivo:");
gets(s);
// Abrir arquivo.
fp = fopen(s, "r");
// Verificar se foi ou não realizada a
// abertura do arquivo.
if (fp == NULL)
printf("Impossível abrir o arquivo %s\n",s);
else
{printf("Arquivo %s aberto com sucesso!!! \n",s);
fclose(fp)};
}
```

Observe que no programa anterior se a abertura falhar, então, será colocado o valor NULL no ponteiro para o arquivo. Se o arquivo puder ser aberto, é colocado em fp o endereço da estrutura criada por **fopen**.

PT2: Reescreva o **PT1** de forma que a operação de abertura de um arquivo seja realiza por uma função e o usuário

possa inserir o nome do arquivo pelo teclado.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MODO "w"

// Apelido para vetor tipo char: cadeia.
typedef char  *cadeia;
// Protótipo de função.
FILE *AbreArquivo(cadeia path, cadeia modo);

main()
{
  FILE *fp;
  cadeia ARQ;
  printf("Digite o nome do arquivo: ");
  gets(ARQ);
  printf("O arquivo e: %s",ARQ);
  fp = AbreArquivo(ARQ,MODO);
  fclose(fp);
  system("PAUSE");
}

FILE *AbreArquivo(cadeia path, cadeia modo)
{
   FILE *arq;
   arq = fopen(path, modo);
   if (arq == NULL)
   {  printf("\n O arquivo %s nao foi criado.",path);
      getch();
      exit(1);
    }
  else
  {
     printf("\n Arquivo %s criado com sucesso.",path);
  }
  return arq;
}
```

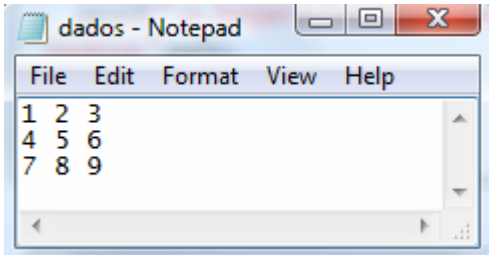
LEITURA E ESCRITA

Para ler e escrever em arquivos tipo texto, ou seja, arquivos onde a informação pode ser visualizada e modificada utilizando-se um editor de textos (por exemplo, o bloco de notas), devem ser usados os comandos de entrada e saída formatada **fscanf** e **fprintf**. Esses comandos funcionam da mesma forma **scanf** e **printf**, tendo apenas mais um parâmetro inicial que indica em qual arquivo o processamento será realizado. Sua sintaxe é dada por:

```
fscanf(FILE *arq, "format", "args");
```

onde: format é a **tag** relativa ao tipo da variável (**%d**, **%f**, **%c**, **%s**) e args são as variáveis cujo conteúdo será manipulado pelos comandos.

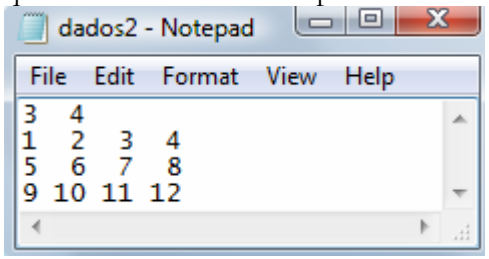
PT3: Escreva um arquivo "dados.txt" no Aplicativo "Notepad" ou "Bloco de notas" uma matriz tal como abaixo:




```
#include <stdio.h>
#include <stdlib.h>
main()
{
  int a[3][3], i, j;
  FILE *arq;
  // Verificando se existe o arquivo !
  if (   (arq=fopen("dados.txt",  "r")) == NULL)
  {
    printf("Erro ao abrir dados.txt \n");
    exit(0); // termina o programa.
  }
  // Leitura dos dados.
  for (i=0; i < 3; i++)
    for (j=0; j < 3; j++)
      fscanf(arq, "%d", &a[i][j]);
  fclose(arq);  // Após ler, fechar arquivo.
}
```

PE1: Modifique o **PT3** para que os valores lidos e armazenados na matriz **a** sejam mostrados na tela.

PT4: Modifique o **PT3** para que o mesmo seja capaz de realizar alocação dinâmica de memória e possa ler os dados do seguinte arquivo "dados2.txt" dado por:



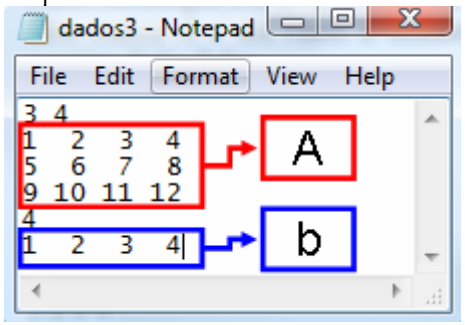
Observe que os dois primeiros valores

correspondem ao número de linhas e colunas da matriz.

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

main()
{
  int **a;
  int m,n, i, j;
  FILE *arq;
  // Verificando se o arquivo existe !
  if ( (arq = fopen("dados2.txt", "r")) == NULL)
  {
    printf("Erro ao abrir dados2.txt \n");
    getch();
    exit(0); // termina o programa.
  }
  // Leitura dos dados do arquivo.
  fscanf(arq,"%d %d",&m,&n);
  // Alocação dinâmica.
  a = (int **) calloc(m, sizeof(int *));
  for (i = 0; i < m; i++)
    a[i] = (int *) calloc(n,sizeof(int));
  for (i = 0; i < m; i++)
    for (j = 0; j <n; j++)
      fscanf(arq,"%d",&a[i][j]);
  // Fechando o arquivo.
  fclose(arq);
  // Impressao na tela.
  printf("A = \n");
  for (i=0; i < m; i++)
  {
    for (j=0; j < n; j++)
      printf(" [%4d] ", a[i][j]);
    printf("\n");
  }
  system("pause");
}
```

PE2: Modifique o **PT4** para ser capaz de ler uma matriz A e um vetor b fornecidos no arquivo dados3.dat.



A partir dos valores da matriz A e do vetor b, calcule e mostre o valor de um vetor x que é obtido por x = A*b.

PE3: Modifique o **PT4** para imprimir o vetor x não somente na tela, mas também em um novo arquivo "dataout.txt".

PE4: Combine o **PE2** e o **PE3** e imprima o valor do vetor x obtido no **PE2** usando o arquivo descrito no **PE3**.

PT5: Criar um programa no qual uma lista de nomes é fornecida pelo usuário através do teclado. Após a leitura, cada nome é gravado em um arquivo lista.txt:

```
#include <stdio.h>
main()
{
  char amigo[80]; char n_arq[80];
  FILE *arq;
  printf("Entre nome do arquivo:");
  gets(n_arq);
  if ( (arq = fopen(n_arq, "w")) == NULL)
  {
  exit(1); // termina execução do progr
  }
  // Laço que lê e armazena os nomes.
  do
  {
    printf("Amigo:");
    gets(amigo);
    fprintf(arq, "%s \n", amigo);
  } while ( *amigo != '\0');
  fclose(arq);  // Fecha arquivo !
}
```

Observe que o programa PT5 lerá os nomes e os escreverá no arquivo lista.txt até que <ENTER> seja pressionado.

PE5: Modifique o PT5 para imprimir na tela cada nome gravado no arquivo.

PE6: Modifique o **PT5** para que a condição de parada de leitura de strings use a condição: **amigo[0] != '\0'**.

Outro comando útil é aquele que verifica se o final de um arquivo foi atingido. O final de um arquivo é indicado através do caractere EOF. O comando para identificar se o final de um arquivo atingido é feof e sua sintaxe é:

```
feof(FILE *arq);
```

PT6: Criar um programa que lê cada letra contida em um arquivo e imprime em outro arquivo todas as letras em maiúsculas (use o comando **toupper** da biblioteca **<ctype.h>**) até encontrar o final do arquivo.

```
#include <stdio.h>
#include <ctype.h>
main()
{
  FILE *in, *out;
  char c;
  in = fopen("entrada.txt", "r");
  out = fopen("saida.txt", "w");
  if ( in == NULL)
  {
    printf("Erro ao criar %s", in);
    exit(1); // termina o programa.
  }
  while ( !feof(in) ) // ate o fim arquivo.
  {
    c = fgetc(in); // lê caractere de in.
    printf("%c",c);
    fputc(toupper(c),out); // escreve em
out
  }
  fclose(in);
  fclose(out);
}
```

Observe que no programa anterior foram utilizados os commandos **fgetc** e **fputc** para leitura e escrita de caracteres em arquivos. A sintaxe e funcionamento destes comandos é semelhante a dos comandos **getc** e **putc** usados para impressão de caracteres na tela.