Linguagem C

Variáveis

Abaixo segue uma tabela com os tipos básicos de variáveis usadas na linguagem C.

TIPO	VALOR ARMAZENADO	INTERVALO	TAMANHO (bytes)
int	números inteiros positivos e negativos	-32.768 a 32.767	2
char	caracteres e números inteiros positivos e negativos	-128 a 127	1
float	números em ponto flutuante positivos e negativos com precisão simples	3.4E-38 a 3.4E+38	4
double	números em ponto flutuante positivos e negativos com precisão dupla	-1.7E-308 a 1.7E+308	8
unsigned int	números inteiros positivos	0 a 65.535	2
unsigned char	caracteres e números inteiros positivos	0 a 255	1
long int	números inteiros positivos e negativos	-2.147.483.648 a 2.147.483.647	4
unsigned long int	números inteiros positivos	0 a 4.292.967.265	4

1.1 Determinando o tamanho de uma variável

Quando você precisar determinar o tamanho de uma variável use o operador sizeof. Ele retorna o número de bytes de uma variável. Sua sintaxe é:

sizeof(VARIÁVEL);

onde VARIÁVEL pode ser uma variável ou um tipo de dado.

Exemplo

```
/* usando o operador sizeof */

#include <stdio.h>

int main()
 {
   int nr;

   printf("A variável nr é um inteiro e tem %d bytes.\n",sizeof(nr));
   printf("Já o tipo de dado char tem %d bytes.\n",sizeof(char));

   return(0);
 }
```

2. Register

Uma variável do tipo register, sempre que possível, é armazenada dentro dos registradores, aumentando a performance do programa. Você deverá usá-las com as variáveis que seu programa deverá acessar repetidamente como as variáveis controladoras de laço.

Exemplo:

```
int main()
 {
   register int contador;
   .....
   .....
   .....
 }
```

3. Interrompendo um laço

Para interromper um laço, seja ele um laço for ou um laço while, você pode usar os comandos continue e break.

O comando continue interrompe o laço e continua na próxima iteração.

O comando break interrompe o laço e continua na próxima instrução de programa após o laço.

4. printf

A função printf

A função printf é parte de um conjunto de funções pré-definidas armazenadas em uma biblioteca padrão de rotinas da linguagem C. Ela permite apresentar na tela os valores de qualquer tipo de dado. Para tanto, printf utiliza o mecanismo de formatação, que permite traduzir a representação interna de variáveis para a representação ASCII que pode ser apresentada na tela.
O primeiro argumento de printf é um string de controle, uma seqüência de caracteres entre aspas. Esta string, que sempre deve estar presente, pode especificar através de caracteres especiais (as seqüências de conversão) quantos outros argumentos estarão presentes nesta invocação da função. Estes outros argumentos serão variáveis cujos valores serão formatados e apresentados na tela. Por exemplo, se o valor de uma variável inteira x é 12, então a execução da função

```
printf("Valor de x = %d", x);
```

imprime na tela a frase Valor de x = 12. Se y é uma variável do tipo caráter com valor 'A', então a execução de

```
printf("x = %d e y = %c\n", x, y);
```

imprime na tela a frase x = 12 e y = A seguida pelo caráter de nova linha (\n), ou seja, a próxima saída para a tela aconteceria na linha seguinte. Observe que a seqüência de conversão pode ocorrer dentro de qualquer posição dentro do *string* de controle.
A função printf não tem um número fixo de argumentos. Em sua forma mais simples, pelo menos um argumento deve estar presente -- a string de controle. Uma string de controle sem nenhuma seqüência de conversão será literalmente impressa na tela. Com variáveis adicionais, a única forma de saber qual o número de variáveis que será apresentado é por inspeção da string de controle. Desta forma, cuidado deve ser tomado para que o número de variáveis após a string de controle esteja de acordo com o número de seqüências de conversão presente na string de controle.
Além de ter o número correto de argumentos e seqüências de conversão, o tipo de cada variável deve estar de acordo com a seqüência de conversão especificada na string de controle. A seqüência de conversão pode ser reconhecida dentro da string de controle por iniciar sempre com o caráter %.
As principais seqüências de conversão para variáveis caracteres e inteiras são:
%c imprime o conteúdo da variável com representação ASCII;
%d imprime o conteúdo da variável com representação decimal com sinal;
%u imprime o conteúdo da variável com representação decimal sem sinal;
%o imprime o conteúdo da variável com representação octal sem sinal;
%x imprime o conteúdo da variável com representação hexadecimal sem sinal.

Uma largura de campo pode ser opcionalmente especificada logo após o caráter %, como em %12d para especificar que o número decimal terá reservado um espaço de doze caracteres para sua representação. Se a largura de campo for negativa, então o número será apresentado alinhado à esquerda ao invés do comportamento padrão de alinhamento à direita. Para a conversão de variáveis do tipo long, o caráter l também deve ser especificado, como em %ld.

Para converter variáveis em ponto flutuante, as seqüências são:
%f
imprime o conteúdo da variável com representação com ponto decimal;
%e imprime o conteúdo da variável com representação em notação científica (exponencial);
%g formato geral, escolhe a representação mais curta entre %f e %e.
Como para a representação inteira, uma largura de campo pode ser especificada para números reais. Por exemplo, %12.3f especifica que a variável será apresentada em um campo de doze caracteres com uma precisão de três dígitos após o ponto decimal.
Finalmente, se a variável a ser apresentada é uma seqüência de caracteres (uma *string*), então o formato de conversão %s pode ser utilizado. Para apresentar o caráter %, a seqüência %% é utilizada.

4.1 Especificadores de formato

ESPECIFICADOR	VALOR
%d	inteiro
%o	inteiro em formato octal
%x	inteiro em formato hexadecimal
%X	
%u	unsigned int
%ld	long int
%f	float
%c	char
%e	float em formato exponencial
%E	
%g	float. C escolhe melhor maneira de exibição entre normal e exponencial
%G	
%s	string

%p	**endereço de um ponteiro**
%n	**quantos caracteres a função printf exibiu**

4.2 Exibindo o sinal de positivo ou negativo antes de um número

Por padrão o sinal de subtração precede um número negativo. Para que o sinal de adição preceda um número positivo inclua um sinal de adição logo após o % no especificador de formato.

Exemplo:

```
#include <stdio.h>

int main()
 {
   int nr_pos,nr_neg;

   nr_pos = 3;
   nr_neg = -3;

   printf("nr_pos = %+d\n",nr_pos);
   printf("nr_neg = %d\n",nr_neg);

   return(0);
 }
```

4.3 Formatando valores inteiros

SINTAXE	EFEITO
printf (" %5d ",valor);	**exibe valor com um mínimo de 5 caracteres**

printf(" %05d ",valor);	**exibe valor com um mínimo de 5 caracteres precedendo-o com zeros**
#%o	**exibe um valor octal precedido de 0 (zero)**
#%x	**exibe um valor hexadecimal precedido de 0x**
#%X	

4.4 Formatando valores float

printf(" %5.3f ", valor); /* Exibe valor com um mínimo de 5 caracteres e com 3 dígitos a direita do ponto decimal */

4.5 Justificando à esquerda

Por padrão, printf justifica o texto à direita. Para justificar à esquerda coloque um sinal de subtração após o %.

Exemplo:

```
#include <stdio.h>

int main()
  {
    int valor = 1;

    printf("Justificado a direita  => %5d\n",valor);

    printf("Justificado a esquerda => %-5d\n",valor);

    return(0);
  }
```

4.6 Quebrando uma string em duas linhas

```
#include <stdio.h>

int main()
  {
    printf("Esta string é muito grande e por isso foi\
 quebrada em mais de uma linha. Para fazer isso você\
 deve usar o sinal de barra invertida.");

    return(0);
  }
```

4.7 Caracteres de escape

CARACTERE	SIGNIFICADO
\a	aviso sonoro
\b	retrocesso
\f	avanço de formulário
\n	nova linha
\r	retorno do carro (sem alimentação de linha)
\t	tabulação horizontal
\v	tabulação vertical
\\	caractere de barra invertida
\'	apóstrofe
\"	aspas
\?	interrogação
\nnn	valor ASCII em octal
\xnnn	valor ASCII em hexadecimal

4.8 Verificando quantos caracteres printf exibiu

```
#include <stdio.h>

int main()
  {
```

```
    int nr_caracteres;

    printf("Verificando quantos caracteres printf exibiu.%n",&nr_caracteres);
    printf("\nA frase acima tem %d caracteres.",nr_caracteres);

    return(0);
  }
```

No exemplo acima, o especificador %n coloca o número de caracteres exibidos por printf na variável &nr_caracteres.

4.9 Usando o controlador ANSI para exibir em cores, limpar a tela e posicionar o cursor

Exibindo em cores

SEQUÊNCIA DE ESCAPE	COR
\033[30m	Cor do primeiro plano preta
\033[31m	Cor do primeiro plano vermelha
\033[32m	Cor do primeiro plano verde
\033[33m	Cor do primeiro plano laranja
\033[34m	Cor do primeiro plano azul
\033[35m	Cor do primeiro plano magenta
\033[36m	Cor do primeiro plano ciano
\033[37m	Cor do primeiro plano branca
\033[40m	Cor do fundo preta
\033[41m	Cor do fundo vermelha

\033[42m	**Cor do fundo verde**
\033[43m	**Cor do fundo laranja**
\033[44m	**Cor do fundo azul**
\033[45m	**Cor do fundo magenta**
\033[46m	**Cor do fundo ciano**
\033[47m	**Cor do fundo branca**

Exemplo :

```
#include <stdio.h>

int main()
  {
    printf("\033[41m"); /* fundo vermelho */
    printf("\033[37m"); /* primeiro plano branco */
    printf("Exibindo o fundo em vermelho e o primeiro plano em branco.\n");
    return(0);
  }
```

Posicionando o cursor

SEQUÊNCIA DE ESCAPE	FUNÇÃO
\033[x;yH	**posiciona o cursor na linha x, coluna y**

\033[xA	**move o cursor x linhas para cima**
\033[xB	**move o cursor x linhas para baixo**
\033[yC	**move o cursor y colunas para a direita**
\033[yD	**move o cursor y colunas para a esquerda**
\033[S	**armazena a posição atual do cursor**
\033[U	**restaura a posição do cursor**
\033[2J	**limpa a tela**
\033[K	**limpa a linha atual**

Exemplo:

```
#include <stdio.h>

int main()
  {
    printf("\033[2J"); /* limpa a tela */
    return(0);
  }
```

5. Outros operadores

operador de incremento => ++

contador++ ==> é o mesmo que contador = contador + 1

contador++ ==> a variável contador é utiliizada e depois incrementada

++contador ==> a variável contador é incrementada e depois utilizada

operador de decremento => --

contador-- ==> é o mesmo que contador = contador - 1

contador-- ==> a variável contador é utiliizada e depois decrementada

--contador ==> a variável contador é decrementada e depois utilizada

simplificando atribuições de expressão a uma variável

EXPRESSÃO NORMAL	EXPRESSÃO SIMPLIFICADA
total = total + 100	total + = 100
conta = conta - 5	conta - = 5
metade = metade / 2	metade / = 2

operador condicional

(CONDIÇÃO) ? COMANDO_V : COMANDO_F;

CONDIÇÃO é avaliada. Se for verdadeira, COMANDO_V será executado. Caso contrário, COMANDO_F será executado.

6. strings

6.1 Determinando o tamanho de uma string

Para determinar o tamanho de uma string use a função **strlen**( ). Esta função faz parte do arquivo de cabeçalho string.h. Sua sintaxe é:

strlen(string)

Exemplo:

```
/* Determinando o tamanho de uma string usando
 * a função strlen() */

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

int main()
  {
    char string[20];

    printf("\n");
```

```
   printf("Determinando o tamanho de uma string\n");
   printf("-----------------------------------\n");
   printf("\n");
   printf("Digite a string :");
   scanf("%s",&string);
   printf("\n");
   printf("A string tem %d carateres.\n\n",strlen(string));
   return(0);
 }
```

6.2 Copiando uma string em outra

Para copiar uma string em outra use a função strcpy( ). Esta função faz parte do arquivo de cabeçalho string.h. Sua sintaxe é:

strcpy(destino, origem)

Exemplo:

```
/* Copiando uma string em outra usando a
 * função strcpy() */

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

int main()
 {
   char string1[10], string2[10];

   printf("\n");
   printf("Copiando uma string em outra\n");
   printf("----------------------------\n");
   printf("\n");
   printf("Digite string1 :");
   scanf("%s",&string1);
   printf("\n");
   printf("string1 = %s\n",string1);
   printf("string2 = %s\n",strcpy(string2,string1));
   return(0);
 }
```

Na prática, todo conteúdo de string2 é substituído por string1.

6.3 Unindo duas strings

Para unir duas strings use a função strcat( ). Esta função faz parte do arquivo de cabeçalho string.h. Sua sintaxe é:

strcat(destino, origem)

Exemplo:

```
/* Unindo duas strings usando a
 * função strcat() */

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

int main()
 {
   char string1[100], string2[10];

   printf("\n");
   printf("Unindo duas strings\n");
   printf("-------------------\n");
   printf("\n");
   printf("Digite string1 :");
   scanf("%s",&string1);
   printf("\n");
   printf("Digite string2 :");
   scanf("%s",&string2);
   printf("\n");
   printf("Unindo string1 a string2 : %s\n\n",strcat(string2,string1));
   return(0);
 }
```

6.4 Anexando caracteres de uma string em outra

Para anexar caracteres de uma string em outra use a função strncat( ). Esta função faz parte do arquivo de cabeçalho string.h. Sua sintaxe é :

strncat(destino, origem, nr_caracteres)

Exemplo:

```
/* Anexando caracteres de uma string
 * em outra usando a função strncat()*/

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

```
int main()
 {
   char string1[20],string2[6]="aeiou";

   printf("\n");
   printf("Anexando caracteres de uma string em outra\n");
   printf("----------------------------------------\n");
   printf("Entre com string1 :");
   scanf("%s",&string1);
   printf("\n");
   printf("string2 = %s\n\n",string2);
   printf("string1 + 3 caracteres de string 2 = %s\n",strncat(string1,string2,3));
   printf("\n");
   return(0);
 }
```

6.5 Função que determina se duas strings são iguais

```
int streql(char *str1, char *str2)
 {
   while((*str1 == *str2) && (*str1))
     {
       str1++;
       str2++;
     }
   return((*str1 == NULL) && (*str2 == NULL));
 }
```

6.6 Convertendo uma string para maiúsculas

Para converter uma string para maiúsculas use a função strupr( ). Esta função faz parte do arquivo de cabeçalho string.h. Sua sintaxe é:

```
strupr(string)
```

Exemplo:

```
/* Convertendo uma string em maiúsculas
 * usando a função strupr() */

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

int main()
 {
```

```
   char string[20];

   printf("\n");
   printf("Convertendo uma string para maiúsculas\n");
   printf("-----------------------------------\n");
   printf("\n");
   printf("Entre com a string :");
   scanf("%s",&string);
   printf("\n");
   printf("string digitada : %s\n",string);
   printf("\n");
   printf("Convertendo para maiúsculas : %s\n",strupr(string));
   return(0);
 }
```

6.7 Convertendo uma string para minúsculas

Para converter uma string para minúsculas use a função strlwr( ). Esta função faz parte do arquivo de cabeçalho string.h. Sua sintaxe é:

strlwr(string)

Exemplo:

```
/* Convertendo uma string em minúsculas
* usando a função strlwr() */

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

int main()
 {
   char string[20];

   printf("\n");
   printf("Convertendo uma string para minúsculas\n");
   printf("-----------------------------------\n");
   printf("\n");
   printf("Entre com a string :");
   scanf("%s",&string);
   printf("\n");
   printf("string digitada : %s\n",string);
   printf("\n");
```

```
   printf("Convertendo para minúsculas : %s\n",strlwr(string));
   return(0);
 }
```

6.8 Localizando a primeira ocorrência de um caractere numa string

Para isso use a função strchr( ). Ela faz parte do arquivo de cabeçalho string.h. Sua sintaxe é:

strchr(string, caracter)

Este função retorna um ponteiro para a primeira ocorrência de "caracter". Caso "caracter" não seja encontrado, ela retornará um ponteiro para o caractere NULL que marca o final da string.

Exemplo:

```
/* Localizando o primeiro caracter numa string
 * usando a função strchr()*/

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

int main()
 {
   char string[30] = "Teste da função strchr().";
   char *ptr;

   printf("\n%s\n",string);

   ptr = strchr(string, 's');

   if (*ptr)
     {
       printf("\n");
       printf("A primeira ocorrência de s é na posição %d\n",ptr - string);
     }
   else
     printf("Caractere não encontrado.\n");

   return(0);
 }
```

6.9 Localizando a última ocorrência de um caractere numa string

Para isso use a função strrchr( ). Ela faz parte do arquivo de cabeçalho string.h. Sua sintaxe é:

strrchr(string, caracter)

Este função retorna um ponteiro para a última ocorrência de "caracter". Caso "caracter" não seja encontrado, ela retornará um ponteiro para o caractere NULL que marca o final da string.

Exemplo:

```
/* Localizando o último caracter numa string
 * usando a função strrchr()*/

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

int main()
 {
   char string[30] = "Teste da função strrchr().";
   char *ptr;

   printf("\n%s\n",string);

   ptr = strrchr(string, 's');

   if (*ptr)
     {
       printf("\n");
       printf("A última ocorrência de s é na posição %d\n",ptr - string);
     }
   else
     printf("Caractere não encontrado.\n");

   return(0);
 }
```

6.10 Função que conta o número de ocorrências de um caractere numa string

```
int contachar(char string[], char letra)
 {
   int contador, tamanho, ocorrencia = 0;
   tamanho = strlen(string);
   for(contador=1;contador <= tamanho;contador++)
     if(string[contador] == letra)
        ocorrencia++;
```

```
   return(ocorrencia);
  }
```

Abaixo segue um exemplo com a utilização da função contachar(). O exemplo considera que ela faz parte do arquivo de cabeçalho <samfunc.h>:

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

int main()
  {
    char *string[20];
    char letra[2];
    int nr;

    printf("Testando a função contachar()\n");
    printf("--------------------------\n");
    printf("\n");
    printf("Entre com a string :");
    scanf("%s",&string);
    printf("\n");
    printf("Entre com a letra :");
    scanf("%s",&letra);
    nr = contachar(string,letra[0]);
    printf("\n");
    printf("Contando o número de ocorrências : %d\n",nr);
    return(0);
  }
```

6.11 Invertendo uma string utilizando a função strrev( )

Para inverter o conteúdo de uma string use a função strrev( ). Ela faz parte do arquivo de cabaçalho string.h. Sua sintaxe é:

```
strrev(string)
```

Exemplo:

```
/* Invertendo uma string usando a
 * função strrev() */

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

```
int main()
  {
    char string[20];

    printf("\n");
    printf("Invertendo uma string\n");
    printf("---------------------\n");
    printf("\n");
    printf("Entre com a string :");
    scanf("%s",&string);
    printf("\n");
    printf("Invertendo ==> %s",strrev(string));
    return(0);
  }
```

6.12 Substituindo os caracteres da string por um único caracter

Para substituir todos os caracteres da string pelo mesmo caracter use a função strset( ). Ela faz parte do arquivo de cabeçalho string.h. Sua sintaxe é:

strset(string,caracter)

Exemplo:

```
/* Substituindo todos os caracteres da string
 * pelo mesmo caracter usando a função strset() */

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

int main()
  {
    char string[20];
    char caracter[2];

    printf("\n");
    printf("Substituindo os caracteres da string\n");
    printf("-----------------------------------\n");
    printf("\n");
    printf("Digite a string :");
    scanf("%s",&string);
    printf("\n");
    printf("Caractere :");
```

```
   scanf("%s",&caracter);
   printf("\n");
   printf("Substituindo ==> %s",strset(string,caracter[0]));
   return(0);
 }
```

6.13 Comparando duas strings

Para comparar duas strings use a função strcmp( ). Ela faz parte do arquivo de cabeçalho string.h. Sua sintaxe é:

strcmp(string1,string2)

Se as strings forem iguais a função retorna zero, se string1 for maior a função retorna um valor menor que zero e se string2 for maior a função retorna um valor maior que zero.

Exemplo:

```
/* Comparando duas strings com a função strcmp() */

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

int main()
 {
   char string1[20],string2[20];
   int retorno;

   printf("\n");
   printf("Comparando duas strings\n");
   printf("----------------------\n");
   printf("\n");
   printf("Entre com a primeira string :");
   scanf("%s",&string1);
   printf("\n");
   printf("Entre com a segunda string :");
   scanf("%s",&string2);
   printf("\n");

   retorno = strcmp(string1,string2);

   if(retorno == 0)
     printf("As strings são iguais.\n");
   else if(retorno < 0)
```

```
    printf("A string1 , maior.\n");
  else
    printf("A string2 , maior.\n");


  return(0);
 }
```

OBSERVAÇÕES:

A função strcmp( ) possui uma variante, a função strncmp( ) que compara os n primeiros caracteres de duas strings. Sua sintaxe é:

strncmp(string1,string2,nr_caracteres)

Existem ainda as funções stricmp( ) e strncmpi( ) que comparam duas strings sem considerar a caixa das letras (maiúsculas ou minúsculas).

6.14 Convertendo strings em números

Para converter strings em números utilize as funções abaixo:

FUNÇÃO	CONVERTE STRINGS EM
atof(string)	**float**
atoi(string)	**int**
atol(string)	**long int**
strtod(string)	**double**
strtol(string)	**long**

Estas funções fazem parte do arquivo de cabeçalho stdlib.h

Exemplo:

```
/* Convertendo strings em números */
#include <stdio.h>
#include <stdlib.h>

int main()
 {
  char string1[20],string2[20];

  printf("\n");
  printf("Convertendo strings em números\n");
```

```
   printf("-----------------------------\n");
   printf("\n");
   printf("Entre com a primeira string :");
   scanf("%s",&string1);
   printf("\n");
   printf("Entre com a segunda string :");
   scanf("%s",&string2);
   printf("\n");
   printf("string1 + string2 = %f",atof(string1) + atof(string2));
   return(0);
 }
```

6.15 Duplicando uma string

Para duplicar uma string use a função strdup( ). Ela está no arquivo de cabeçalho string.h. Sua sintaxe é:

*strdup(string)

Exemplo:

```
/* Duplicando uma string */

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

int main()
 {
   char string[20];
   char *copia;

   printf("\n");
   printf("Duplicando uma string\n");
   printf("---------------------\n");
   printf("\n");
   printf("Entre com a string :");
   scanf("%s",&string);
   printf("\n");
   copia = strdup(string);
   printf("string ==> %s\n",string);
   printf("cópia  ==> %s\n",copia);
   return(0);
 }
```

6.16 Localizando uma substring dentro da string

Para localizar uma substring dentro da string use a função strstr( ). Ela pertence ao arquivo de cabeçalho string.h e sua sintaxe é:

strstr(string,substring)

Se a substring existir dentro da string, a função retornará um ponteiro para a primeira letra da substring, senão retornará NULL.

Exemplo:

```
/* Localizando uma substring dentro de uma string */

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

int main()
 {
   char string[20],substring[20];
   char *extrai;
   int tamanho;

   printf("\n");
   printf("Localizando uma substring dentro da string\n");
   printf("----------------------------------------\n");
   printf("\n");
   printf("Entre com a string :");
   scanf("%s",&string);
   printf("\n");
   printf("Entre com a substring :");
   scanf("%s",&substring);
   tamanho = strlen(substring);
   extrai = strstr(string,substring);
   printf("\n");

   if(extrai)
    {
      printf("A string contém a substring.\n");
      printf("A substring começa na posição %d.\n",extrai-string);
      printf("A substring tem %d caracteres.\n",tamanho);
    }
   else
```

```
      printf("A string não contém a substring.\n");

    return(0);
  }
```

6.17 Função que remove uma substring de dentro de uma string

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

char *sstr(char *string, char *substring)
  {
    char *extrai;
    int tamanho,contador;

    tamanho = strlen(substring);
    extrai = strstr(string,substring);

    if(extrai)
      {
        for(contador = 0;contador < tamanho; contador++)
          extrai[contador] = string[(extrai - string) + contador];
        extrai[contador] = NULL;
        return(extrai);
      }
    else
      return(" ");
  }

int main()
  {
    char string[20],substring[20];

    printf("\n");
    printf("Entre com a string :");
    scanf("%s",&string);
    printf("\n");
    printf("Entre com a substring :");
    scanf("%s",&substring);
    printf("\n");
```

```
   printf("substring ==> %s\n",sstr(string,substring));

   return(0);
  }
```

6.18 Função que substitui uma subtring por outra

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

char *subs_str(char *string, char *substring, char *nova)
 {
   char *extrai;
   int tamanho1,tamanho2,contador;

   tamanho1 = strlen(substring);
   tamanho2 = strlen(nova);

   if((tamanho1 > tamanho2) || (tamanho2 > tamanho1))
     return(" ");
   else
     {
       extrai = strstr(string,substring);

       if(extrai)
         {
           for(contador = 0;contador < tamanho1; contador++)
             string[(extrai - string) + contador] = nova[contador];
           return(string);
         }
       else
        return(" ");
      }
 }

int main()
 {
   char string[20],substring[20],nova[20];

   printf("\n");
```

```
   printf("Entre com a string :");
   scanf("%s",&string);
   printf("\n");
   printf("Entre com a substring :");
   scanf("%s",&substring);
   printf("\n");
   printf("Entre com a nova substring :");
   scanf("%s",&nova);
   printf("\n");
   printf("nova string ==> %s\n",subs_str(string,substring,nova));

   return(0);
  }
```

6.19 Invertendo uma string sem o uso da função strrev( )

```
/* Invertendo uma string */

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

int main()
  {
   char string[100],invertida[100];
   char *caracter;
   int tamanho,contador;

   printf("\n");
   printf("Invertendo uma string\n");
   printf("---------------------\n");
   printf("\n");
   printf("Entre com a string :");
   scanf("%s",&string);

   tamanho = strlen(string);
   contador = tamanho;

   caracter = &string;

   while(*caracter)
```

```
    {
      invertida[(contador - 1)] = *caracter;
      *(caracter++);
      contador--;
    }
  invertida[tamanho] = NULL;

  printf("\n");
  printf("Invertendo ==> %s\n\n",invertida);
  return(0);
 }
```

7. Caracter

7.1 Verificando se o caracter é uma letra

Para fazer esta verificação utilize a macro isalpha( ). Ela faz parte do arquivo de cabeçalho ctype.h. Sua sintaxe é:

isalpha(caracter)

Exemplo:

```
/* Verificando se um caracter é uma letra
 * usando a macro isalpha() */

#include <stdio.h>
#include <ctype.h>

int main()
 {
  char caracter;

  printf("Digite um caracter :");
  caracter = getchar();
  printf("\n");
  if (isalpha(caracter))
    printf("O caracter é uma letra.\n");
  else
    printf("O caracter não é uma  letra.\n");
  return(0);
 }
```

7.2 Verificando se o caracter é um valor ASCII

Um valor ASCII é um valor entre 0 e 127. Para verificar se um caractere é um valor ASCII utilize a macro isascii( ) que faz parte do arquivo de cabeçalho ctype.h. Sua sintaxe é:

isascii(caracter)

Exemplo :

```
/* Verificando se um caracter contém um valor ASCII
 * usando a macro isascii() */

#include <stdio.h>
#include <ctype.h>

int main()
 {
   char caracter;

   printf("\n");
   printf("Digite um caracter :");
   caracter = getchar();
   printf("\n");
   if (isascii(caracter))
     printf("O caracter contém o valor ASCII %d.\n",caracter);
   else
     printf("O caracter não contém um valor ASCII.\n");

   printf("\n");
   return(0);
 }
```

7.3 Verificando se o caracter é um caracter de controle

Um caracter de controle é composto pelo pressionamento da tecla control (CTRL) e uma letra (^A , ^B, ^C, ... ^Z ou ^a, ^b, ^c ..... ^z). Para verificar se um caractere é de controle use a macro iscntrl( ). Ela faz parte do arquivo de cabeçalho ctype.h e sua sintaxe é:

iscntrl(caracter)

Exemplo:

```
/* Verificando se um caracter é de controle
 * usando a macro iscntrl() */
```

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

int main()
 {
   char caracter;

   printf("\n");
   printf("Digite um caracter :");
   caracter = getchar();
   printf("\n");
   if (iscntrl(caracter))
    {
      printf("O caracter digitado é um caracter de controle\n");
      printf("e equivale ao código ASCII %d.\n",caracter);
    }
   else
     printf("O caracter digitado não é um caracter de controle.\n");

   printf("\n");
   return(0);
 }
```

7.4 Verificando se o caracter é um dígito

Para verificar se o caracter é um dígito use a macro isdigit( ). Ela pertence ao arquivo de cabeçalho ctype.h e sua sintaxe é:

```
isdigit(caracter)
```

Exemplo :

```
/* Verificando se um caracter é um dígito
 * usando a macro isdigit() */

#include <stdio.h>
#include <ctype.h>

int main()
 {
   char caracter;

   printf("\n");
```

```
   printf("Digite um caracter :");
   caracter = getchar();
   printf("\n");
   if (isdigit(caracter))
     printf("O caracter é um dígito.\n");
   else
     printf("O caracter não é um dígito.\n");

   printf("\n");
   return(0);
 }
```

7.5 Verificando se o caracter é maiúsculo

Para verificar se o caracter é maiúsculo use a macro isupper( ). Ela pertence ao arquivo de cabeçalho ctype.h e sua sintaxe é:

```
isupper(caractere)
```

Exemplo:

```
/* Verificando se um caracter é maiúsculo
 * usando a macro isupper() */

#include <stdio.h>
#include <ctype.h>

int main()
 {
   char caracter;

   printf("\n");
   printf("Digite um caracter :");
   caracter = getchar();
   printf("\n");
   if (isupper(caracter))
     printf("O caracter é maiúsculo.\n");
   else
     printf("O caracter não é um maiúsculo.\n");

   printf("\n");
   return(0);
 }
```

7.6 Verificando se o caracter é minúsculo

Para verificar se o caracter é minúsculo use a macro islower( ). Ela pertence ao arquivo de cabeçalho ctype.h e sua sintaxe é:

islower(caractere)

Exemplo:

```
/* Verificando se um caracter é minúsculo
 * usando a macro islower() */

#include <stdio.h>
#include <ctype.h>

int main()
 {
   char caracter;

   printf("\n");
   printf("Digite um caracter :");
   caracter = getchar();
   printf("\n");
   if (islower(caracter))
     printf("O caracter é minúsculo.\n");
   else
     printf("O caracter não é um minúsculo.\n");

   printf("\n");
   return(0);
 }
```

7.7 Convertendo um caracter para maiúsculo

Para executar esta conversão você pode usar a macro _toupper( ) ou a função toupper( ). As duas estão no arquivo de cabeçalho ctype.h. A diferença entre as duas é que a macro não testa se o caractere a ser convertido é um minúsculo. Assim, se o caracter não for uma letra minúscula a macro fará uma conversão errada. Se você tiver certeza que o caractere é uma letra minúscula use a macro que é mais rápida, caso contrário use a função. A sintaxe das duas segue abaixo:

_toupper(caracter)

toupper(caracter)

Exemplo :

```
/* Convertendo um caracter para maiúsculo */

#include <stdio.h>
#include <ctype.h>

int main()
 {
   char caracter;

   printf("\n");
   printf("Digite um caracter minúsculo, um dígito ou um símbolo qualquer :");
   caracter = getchar();
   printf("\n");
   printf("Convertendo com a função toupper( ) ==> %c\n",toupper(caracter));
   printf("\n");
   printf("Convertendo com a macro _toupper( ) ==> %c\n",_toupper(caracter));
   printf("\n");
   return(0);
 }
```

7.8 Convertendo um caracter para minúsculo

Para executar esta conversão você pode usar a macro _tolower( ) ou a função tolower( ). As duas estão no arquivo de cabeçalho ctype.h. A diferença entre as duas é que a macro não testa se o caractere a ser convertido é um maiúsculo. Assim, se o caracter não for uma letra maiúscula a macro fará uma conversão errada. Se você tiver certeza que o caractere é uma letra maiúscula use a macro que é mais rápida, caso contrário use a função. A sintaxe das duas segue abaixo:

```
_tolower(caracter)

tolower(caracter)
```

Exemplo :

```
/* Convertendo um caracter para minúsculo */

#include <stdio.h>
#include <ctype.h>

int main()
 {
   char caracter;
```

```
   printf("\n");
   printf("Digite um caracter maiúsculo, um dígito ou um símbolo qualquer :");
   caracter = getchar();
   printf("\n");
   printf("Convertendo com a função tolower( ) ==> %c\n",tolower(caracter));
   printf("\n");
   printf("Convertendo com a macro _tolower( ) ==> %c\n",_tolower(caracter));
   printf("\n");
   return(0);
 }
```

8. Funções

8.1 Forma geral de uma função

```
TIPO NOME(PARÂMETROS)
 {
    CORPO DA FUNÇÃO
 }
```

Onde:

TIPO é o tipo de valor retornado pela função. Se nada for especificado o compilador considera que será retornado um valor inteiro.

NOME é o nome da função

PARÂMETROS é a lista das variáveis que recebem os argumentos quando a função é chamada. Deve incluir o tipo e nome de cada variável. Sua sintaxe é: (tipo variável1, tipo variável2, ..., tipo variáveln)

CORPO é onde estão as instruções da função

Exemplo:

```
int soma(int a,int b)
 {
   int resultado;

   resultado = a + b;

   return(resultado);
 }
```

8.2 Variáveis em funções

As variáveis criadas numa função são locais, assim serão destruídas após o término da função.

Caso você queira manter o valor de uma variável entre as chamadas a uma função você deve declarar esta variável como static. Exemplo:

```
/* Mantendo o valor de uma variável entre as
 * chamadas de uma função */

#include <stdio.h>

int soma_1(int a);

int main()
 {
   int nr = 1;

   printf("Chamando a função a primeira vez: valor + 1 = %d\n",soma_1(nr));
   printf("Chamando a função pela segunda vez: : valor + 1 =
%d\n",soma_1(nr));
   printf("Chamando a função pela terceira vez: : valor + 1 = %d\n",soma_1(nr));

   return(0);
 }


int soma_1(int a)
 {
   static int valor = 1;

   printf("valor = %d\n",valor);

   valor = valor + a;

   return(valor);
 }
```

Caso uma variável local a função tenha o mesmo nome de uma variável global, a variável local será usada e não a global. Exemplo:

```
/* Entre variáveis locais e globais com o mesmo nome
 * dentro de uma função, a variável local é escolhida */

#include <stdio.h>
```

```
int a = 1; /* variável global */

void exibe(void)
 {
   int a = 10; /* variável local a função exibe() */
   printf("a dentro da função = %d\n",a);
 }

int main()
 {
   printf("\n");
   printf("a dentro de main = %d\n",a);
   exibe();
   printf("\n");
   return(0);
 }
```

BIZÚ: Evite variáveis globais.

8.3 Argumentos e parâmetros

Argumentos são os valores usados para chamar a função e parâmetros são as variáveis, declaradas na definição da função, que recebem estes argumentos. Observe o exemplo abaixo:

```
/* Argumentos e parâmetros */

#include <stdio.h>

int soma(int a, int b) /* "a" e "b" são os parâmetros da função "soma" */
 {
   int resultado;
   resultado = a + b;
   return(resultado);
 }

int main()
 {
   printf("A soma entre 5 e 2 é %d\n",soma(5,2));
```

```
  /* No comando printf acima a função "soma" é chamada
   * com os argumentos 5 e 2 */


  return(0);
}
```

Os tipos dos argumentos devem ser compatíveis com os tipos dos parâmetros.

Você encontrará também referência aos *parâmetros formais* e *parâmetros reais*. Os parâmetros formais são os parâmetros propriamente ditos, enquanto que os parâmetros reais são os argumentos.

8.4 Parâmetros

Existem duas formas de declaração de parâmetros em funções: a forma clássica e a forma moderna.

A forma clássica tem a seguinte sintaxe:

```
TIPO NOME(PARÂMETRO1, PARÂMETRO2, ... , PARÂMETROn)
 TIPO DO PARÂMETRO1;
 TIPO DO PARÂMETRO2;
 ...
 ...
 TIPO DO PARÂMETROn;
 {
   CORPO DA FUNÇÃO
 }
```

Já a forma moderna tem a seguinte sintaxe:

```
TIPO NOME(TIPO PARÂMETRO1, TIPO PARÂMETRO2, ... , TIPO
PARÂMETROn)
 {
  CORPO DA FUNÇÃO
 }
```

Abaixo segue um exemplo de função com os dois tipos de declaração de parâmetros.

```
/* Com declaração clássica */
int soma(a, b)
 int a;
 int b;
 {
  int resultado;
  resultado = a + b;
  return(resultado);
 }
```

```
/* Com declaração moderna */
int soma(int a, int b)
 {
   int resultado;
   resultado = a + b;
   return(resultado);
 }
```

Atualmente utiliza-se a forma moderna, porém, em programas mais antigos você encontrará a forma clássica.

8.5 Chamada por valor e chamada por referência

A chamada por valor é a passagem normal do valor dos argumentos para a função. Utilizando esta chamada os valores dos argumentos passados não são modificados. Na realidade é passada uma cópia dos valores para a função.

Na chamada por referência são passados os endereços de memória onde estão os argumentos. Neste tipo de chamada os valores podem ser modificados.

Abaixo segue um exemplo de uma chamada por valor:

```
/* Testando a chamada por valor */

#include <stdio.h>


/* Função com chamada por valor */
int valor(int a, int b)
 {
   a = a + 3; /* Modificando o primeiro argumento */
   b = b + 2; /* Modificando o segundo argumento */

   printf("Valores modificados dentro da função:\n");
   printf("nr1 = %d\n",a);
   printf("nr2 = %d\n",b);
 }


int main()
 {
```

```
    int nr1 = 2, nr2 = 3, total;

    printf("\n");
    printf("Chamada por valor\n");
    printf("=================\n");
    printf("Valores iniciais de nr1 e nr2\n");
    printf("nr1 = %d\n",nr1);
    printf("nr2 = %d\n",nr2);
    printf("\n\nChamando a função\n");
    valor(nr1,nr2); /* Neste tipo de chamada são passados os argumentos
                     * normalmente. Na verdade a função recebe uma cópia
                     * destes argumentos */

    printf("\n\nValores após a chamada da função\n");
    printf("nr1 = %d\n",nr1);
    printf("nr2 = %d\n",nr2);

    return(0);
}
```

Agora o mesmo exemplo com a chamada por referência:

```
/* Testando a chamada por referência */

#include <stdio.h>



/* Função com chamada por referência */
int valor(int *a, int *b)
  {
    *a = *a + 3; /* Modificando o primeiro argumento */
    *b = *b + 2; /* Modificando o segundo argumento */

    printf("Valores modificados dentro da função:\n");
    printf("nr1 = %d\n",*a);
    printf("nr2 = %d\n",*b);
  }


int main()
```

```
 {
   int nr1 = 2, nr2 = 3, total;

   printf("\n");
   printf("Chamada por referência\n");
   printf("======================\n");
   printf("Valores iniciais de nr1 e nr2\n");
   printf("nr1 = %d\n",nr1);
   printf("nr2 = %d\n",nr2);
   valor(&nr1,&nr2); /* Neste tipo de chamada é passado o endereço do
                      * argumento. Neste tipo de chamada os valores
                      * podem ser modificados */

   printf("\n\nValores após a chamada da função\n");
   printf("nr1 = %d\n",nr1);
   printf("nr2 = %d\n",nr2);

   return(0);
}
```

OBSERVAÇÃO: As strings e matrizes sempre são chamadas por referência. Quando C passa uma matriz ou string para uma função é passado o endereço inicial da matriz ou função.

8.6 Argumentos da linha de comando

Caso queira, você pode passar argumentos diretamente para a função main(). Como main () é a primeira função a ser chamada quando você chama o programa os argumentos para ela são passados junto com o comando que chama o programa, geralmente seu nome. Estes argumentos são conhecidos como argumentos da linha de comando. Observe o exemplo abaixo:

```
#include <stdio.h>


int main(int argc, char *argv[])
 {
   printf("Olá %s.\n",argv[1]);


   return(0);
 }
```

Os argumentos da linha de comando são argc e argv. argc armazena o número de argumentos passados para o programa, inclusive o nome do programa. argv é uma matriz de strings e armazena o nome do programa e os argumentos passados. argv[0] armazena o nome do programa, argv[1] armazena o primeiro argumento passado para o programa, argv[2] armazena o segundo argumento

passado para o programa, e assim por diante. Os argumentos são seprarados por um espaço.

8.7 O comando return

O comando return é usado para encerrar a função e retornar um valor para a função chamadora. Exemplo:

```
#include <stdio.h>


float total(float preco, float taxa_juros)
 {
   float preco_final, juros;
   juros  = preco * (taxa_juros / 100);
   preco_final =  preco + juros;
   return(preco_final);
 }

int main()
 {
   float preco, taxa_juros, preco_final;
   printf("\n");
   printf("Preço na etiqueta :");
   scanf("%f",&preco);
   printf("\n");
   printf("Taxa de juros :");
   scanf("%f",&taxa_juros);

   preco_final = total(preco, taxa_juros);


   printf("\n");
   printf("Total a pagar : %4.2f\n\n",preco_final);


   return(0);
 }
```

O valor retornado por return deve ser compatível com o tipo da função, o qual é definido quando da sua declaração.

```
float total(float preco, float taxa_juros)
```

No exemplo, a função total retorna um valor float. Isto é determinado pela colocação do tipo float antes do nome da função, como mostrado acima.

Se uma função não retornar nenhum valor ela é do tipo void. Exemplo:

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

```
void nao_retorna()
 {
   printf("Esta função não retorna nada.\n");
 }

int main()
 {
   nao_retorna();
   return(0);
 }
```

De acordo com o padrão ANSI, a função `main` devolve um inteiro para o processo chamador, que geralmente é o sistema operacional. Isto é equivalente a chamar `exit` com o mesmo valor. Alguns compiladores ainda aceitam que `main` seja declarada como `void` caso não retorne nenhum valor.

8.8 Protótipo de função

A chamada a uma função deve vir, a princípio, após sua definição para o compilador conhecer os tipos de parâmetros e o tipo de retorno da função. Porém, você pode chamar a função antes da definição desta. Para isso declare apenas um protótipo da função, o qual tem apenas o valor de retorno e os parâmetros da função. Observe o exemplo abaixo:

```
#include <stdio.h>

int soma(int a, int b); /* protótipo da função */

int main()
 {
   int nr1, nr2;
   printf("Entre com o primeiro número :");
   scanf("%d",&nr1);
   printf("Entre com o segundo número :");
   scanf("%d",&nr2);
   printf("\n%d + %d = %d\n\n",nr1,nr2,soma(nr1,nr2));
   return(0);
 }

int soma(int a, int b) /* função propriamente dita */
 {
   int resultado;
   resultado = a + b;
```

```
   return(resultado);
 }
```

8.9 Recursão

Recursão é o ato de uma função chamar ela mesma. Uma função que chama a ela mesma é chamada função recursiva.

O exemplo padrão de função recursiva é uma função que calcula o fatorial de um número. O fatorial de um número é igual ao produto dos números inteiros de 1 até o número.

fatorial de 5 = 5 * 4 * 3 * 2 * 1

Se você observar com cuidado verá que:

fatorial de 5 = 5 * fatorial de 4

fatorial de 4 = 4 * fatorial de 3

fatorial de 3 = 3 * fatorial de 2

fatorial de 2 = 2 * fatorial de 1

Ou seja

fatorial de um número = número * (fatorial de número - 1)

Desta conclusão podemos escrever nosso exemplo de função recursiva:

```
/* Exemplo de função recursiva */

#include <stdio.h>

int fatorial(nr)
 {
    int resposta;

    if(nr == 1)
      return(1);
    resposta = nr * fatorial(nr-1);
    return(resposta);
 }



int main()
 {
   int a;
   printf("\nEntre com um valor inteiro :");
   scanf("%d",&a);
   printf("O fatorial de %d é %d\n\n",a,fatorial(a));
   return(0);
```

```
}
```

A recursão sempre deve ser evitada basicamente por dois fatores. Primeiro que uma função recursiva é difícil de compreender. Segundo que as funções recursivas são mais lentas que suas correspondentes não recursivas.

Normalmente uma função recursiva também pode ser escrita com laços de repetição tipo for ou while de modo a remover a recursão.

9. E/S (Entrada/Saída)

9.1 Lendo um caracter do teclado

Para ler um caracter do teclado utilize a função getchar( ). Ela faz parte do arquivo de cabeçalho stdio.h. Sua utilização é:

variavel = getchar();

Esta função retorna o valor inteiro referente ao código ASCII do caractere lido, porém você pode atribuir este valor a uma variável do tipo caracter. Caso ocorra um erro, ela retorna EOF.

Abaixo segue um exemplo da utilização de getchar:

```
/* Exemplo da utilização de getchar */


#include <stdio.h>


int main()
 {
   char caracter;


   printf("\n");
   printf("Utilizando getchar()\n");
   printf("--------------------\n");
   printf("\n");
   printf("Entre com um caracter :");


   caracter = getchar();


   printf("\nVocê digitou o caracter %c\n\n",caracter);


   return(0);
 }
```

9.2 Exibindo um caracter

Para exibir um caracter você pode usar a função putchar( ) que está no arquivo de cabeçalho stdio.h. Sua sintaxe é:

putchar(variavel)

Onde variavel é um número inteiro, porém você pode passar variavel como um caracter. putchar retorna o caracter exibido ou EOF, caso ocorra algum erro. Exemplo:

```
/* Exemplo da utilização de putchar */


#include <stdio.h>


int main()
 {
   char caracter;

   printf("\n");
   printf("Utilizando putchar()\n");
   printf("--------------------\n");
   printf("\n");
   printf("Entre com um caracter :");

   caracter = getchar();

   printf("\nExibindo o caracter com putchar => ");

   putchar(caracter);

   printf("\n\n");

   return(0);
 }
```

9.3 Lendo uma string do teclado

Você pode ler uma string do teclado usando as funções gets( ) e fgets(). Elas fazem parte do arquivo de cabeçalho stdio.h.

O gcc desencoraja o uso de gets . A própria man page de gets declara o seguinte em sua seção PROBLEMAS :

PROBLEMAS

> Nunca use gets(). Porque é impossível saber, sem conhecer antecipada mente os dados, quantos caracteres gets() vai ler, e porque gets() vai continuar a guardar caracteres ultrapassado o fim do 'buffer', ela é extremamente perigosa de usar. Este comportamento tem sido utilizado para quebrar a segurança de computadores. Use fgets() no seu lugar.

Por isso que só abordarei a sintaxe de fgets, que é a seguinte:

```
fgets(STRING,TAMANHO,STREAM);
```

onde:

STRING é a variável onde a string será armazenada

TAMANHO é o tamanho máximo da string

STREAM é de onde os caracteres serão lidos, para ler do teclado o valor padrão para isto é stdin

Exemplo do uso de fgets:

```
/* usando fgets para ler uma string do teclado */

#include <stdio.h>

int main()
 {
   char frase[50];

   printf("Digite uma frase qualquer:");
   fgets(frase,50,stdin);
   printf("\n");
   printf("Exibindo\n\n");

   printf("%s\n",frase);

   return(0);
 }
```

9.4 Exibindo uma string

Você pode exibir uma string usando a função printf ou a função puts( ). Elas fazem parte do arquivo de cabeçalho stdio.h.

A sintaxe de printf para a exibir uma string é;

```
printf("%s",STRING);
```

Exemplo:

```
#include <stdio.h>

int main()
 {
   char string[30];

   printf("\n");
```

```
   printf("Exemplo do uso de printf para exibir strings\n");
   printf("-------------------------------------------\n");
   printf("Digite uma string :");
   fgets(string,30,stdin);
   printf("\n");
   printf("A string digitada foi :%s",string);
   printf("\n\n");
   return(0);
 }
```

A sintaxe de puts é:

```
puts(string)
```

Exemplo:

```
#include <stdio.h>

int main()
 {
   char string[30];

   printf("\n");
   printf("Exemplo do uso de puts\n");
   printf("----------------------\n");
   printf("\n");
   printf("Digite uma string :");
   fgets(string,30,stdin);
   printf("\n");
   printf("A string digitada foi :",string);
   puts(string);
   printf("\n\n");
   return(0);
 }
```

9.5 Saída formatada (printf)

A saída formatada é feita utilizando a função printf vista anteriormente. printf faz parte do arquivo de cabeçalho stdio.h

9.6 Entrada formatada (scanf)

A entrada formatada é feita utilizando a função scanf.Ela faz parte do arquivo de cabeçalho stdio.h. Sua sintaxe é:

```
scanf("especificador de formato",&variável)
```

O especificador de formato segue a mesma sintaxe da função printf.

Observe que o valor entrado é passado para o endereço da variável. No caso de leitura de uma string não há necessidade do operador &, já que o nome de uma string sem o índice é entendido pela linguagem C como um ponteiro para o início da string.

Abaixo segue um exemplo do uso de scanf:

```
# include<stdio.h>

int main()
 {
   int qde;
   float preco,total;
   char produto[20];

   printf("\n");
   printf("Produto    :");
   scanf("%s",produto);
   printf("\n");
   printf("Preço      :");
   scanf("%f",&preco);
   printf("\n");
   printf("Quantidade :");
   scanf("%d",&qde);
   printf("\n");
   printf("Produto\tPreço\tQde\tTotal\n");
   printf("%s\t%.2f\t%d\t%.2f\n\n",produto,preco,qde,qde*preco);
   return(0);
 }
```

10. Funções matemáticas

10.1 Obtendo o valor absoluto de um número inteiro

Para obter o valor absoluto de um número inteiro use a função abs( ). Ela faz parte do arquivo de cabaçalho stdlib.h e sua sintaxe é:

```
abs(número)
```

Exemplo do uso da função abs:

```
#include <stdio.h>

int main()
 {
```

```
    int nr1 = 5,nr2 = -7;

    printf("\n");
    printf("nr1=%d\tabs(nr1)=%d\n",nr1,abs(nr1));
    printf("nr2=%d\tabs(nr2)=%d\n\n",nr2,abs(nr2));
    return(0);
  }
```

10.2 Funções trigonométricas

O arquivo de cabeçalho tgmath.h (ou math.h, dependendo do seu compilador) fornece as seguintes funções trigonométricas:

sin(angulo)

cos(angulo)

tan(angulo)

Estas calculam o seno, co-seno e tangenge de angulo. Todas recebem e retornam um valor double. O argumento angulo passado para as funções é especificado em radianos.

Além destas devem ser encontradas as funções:

asin(angulo)

acos(angulo)

atan(angulo)

Para o cálculo de arco seno, arco co-seno e arco tangente e:

sinh(angulo)

cosh(angulo)

tanh(angulo)

Para o cálculo do seno, co-seno e tangente hiperbólicos. Para maiores detalhes dê uma estudada no arquivo de cabeçalho referente na biblioteca de seu compilador.

10.3 Gerando números aleatórios

Para gerar números aleatórios alguns compiladores possuem em seu arquivo de cabeçalho stdlib.h as funções random( ) e rand( ).

A função random gera um número aleatório entre zero e um número inteiro passado como argumento. Sua sintaxe é:

random(número)

Já a função rand gera um número aleatório entre zero e RAND_MAX que é definido no próprio arquivo stdlib.h. A sintaxe de rand é:

rand( )

Abaixo segue um exemplo do uso destas funções:

```
/* gerando números aleatórios */

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

int main()
 {
   int contador;

   printf("Gerando 5 números aleatórios com random\n");
   for(contador=1;contador <= 5; contador++)
     printf("%d\n",random(10));



   printf("Gerando 5 números aleatórios com rand\n");
   for(contador=1;contador <= 5; contador++)
     printf("%d\n",rand());

   return(0);
 }
```

Porém, ao executar o programa acima várias vezes, você verá que ele sempre gera os mesmos números aleatórios. Para resolver isso basta utilizar as funções randomize( ) e srand( ) que iniciam o gerador de números aleatórios.

randomize inicia o gerador de números aleatórios usando o relógio do computador para produzir uma semente aleatória e srand( ) lhe permite especificar o valor inicial do gerador de números aleatórios. Sua sintaxe é:

srand(número)

Veja como fica nosso exemplo inicial usando randomize para iniciar o gerador de números aleatórios:

```
/* gerando números aleatórios */

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

int main()
```

```
 {
   int contador;

   randomize();
   printf("Gerando 5 números aleatórios com random\n");
   for(contador=1;contador <= 5; contador++)
     printf("%d\n",random(10));



   printf("Gerando 5 números aleatórios com rand\n");
   for(contador=1;contador <= 5; contador++)
     printf("%d\n",rand());

   return(0);
 }
```

Até aqui tudo bem. Porém ao utilizar este código no gcc obtive uma mensagem de erro dizendo que a função random tinha muitos argumentos. Dei uma olhada no arquivo de cabeçalho stdlib.h e vi que neste compilador as coisas funcionam de maneira um pouco diferente.

random e rand aparentemente tem a mesma função, ou seja, geram números aleatórios entre zero e RAND_MAX.

RAND_MAX é definido como igual a 2147483647. Assim eu dividi o resultado de random por 100000000 e consegui alguns aleatórios com dois dígitos:

```
/* gerando números aleatórios no gcc */

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

/* random() gera um long int entre 0 e RAND_MAX = 2147483647
 *
 * RAND_MAX é definido em stdlib.h */

int main()
 {
   int contador;

   printf("Gerando 5 números aleatórios com random\n");
   for(contador=1;contador <= 5; contador++)
     printf("%d\n",random()/100000000);
```

```
    printf("Gerando 5 números aleatórios com rand\n");
   for(contador=1;contador <= 5; contador++)
     printf("%d\n",rand()/100000000);

   return(0);
 }
```

Para iniciar o gerador de números aleatórios a função randomize não existe, porém, a função srandom existe. Então eu a utilizei juntamente com a função time e consegui gerar números aleatórios diferentes a cada execução do programa.

O código anterior ficou assim:

```
/* gerando números aleatórios no gcc */

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

/* random() gera um long int entre 0 e RAND_MAX = 2147483647
 *
 * RAND_MAX é definido em stdlib.h */

int main()
 {
   int contador;

   srandom(time(NULL)); /* iniciando o gerador de números aleatórios */
   printf("Gerando 5 números aleatórios com random\n");
   for(contador=1;contador <= 5; contador++)
     printf("%d\n",random()/100000000);



    printf("Gerando 5 números aleatórios com rand\n");
   for(contador=1;contador <= 5; contador++)
     printf("%d\n",rand()/100000000);

   return(0);
 }
```

11. Arquivos

11.1 Introdução

O sistema de E/S de C utiliza o conceito de streams e arquivos. Uma stream é um dispositivo lógico que representa um arquivo ou dispositivo. A stream é independente do arquivo ou dispositivo. Devido a isso, a função que manipula uma stream pode escrever tanto em um arquivo no disco quanto em algum outro dispositivo, como o monitor.

Existem dois tipos de streams: de texto e binária.

Em uma stream de texto podem ocorrer certas traduções de acordo com o sistema hospedeiro. Por exemplo, um caracter de nova linha pode ser convertido para os caracteres retorno de carro e alimentação de linha. Devido a isso pode não haver uma correspondência entre os caracteres da stream e do dispositivo externo; a quantidade de caracteres pode não ser a mesma.

A stream binária é uma sequência de bytes com uma correspondência de um para um com os bytes encontrados no dispositivo externo, isto é, não ocorre nenhuma tradução de caracteres. O número de bytes é o mesmo do dispositivo.

Um arquivo é interpretado pela linguagem C como qualquer dispositivo, desde um arquivo em disco até um terminal ou uma impressora. Para utilizar um arquivo você deve associá-lo a uma stream e, então, manipular a stream. Você associa um arquivo a uma stream através de uma operação de abertura.

Nem todos os arquivos tem os mesmos recursos. Por exemplo, um arquivo em disco pode suportar acesso aleatório enquanto um teclado não.

Do que foi até aqui exposto concluímos que todas as streams são iguais, mas não todos os arquivos.

Se o arquivo suporta acesso aleatório, abrí-lo inicializa o indicador de posição apontando para o começo do arquivo. Quando cada caracter é lido ou escrito no arquivo, o indicador de posição é incrementado.

Um arquivo é desassociado de uma stream através de uma operação de fechamento. Se um arquivo aberto para saída por fechado, o conteúdo de sua stream será escrito no dispositivo externo. Esse processo é geralmente chamado de descarga (flushing) da stream e garante que nenhuma informação seja acidentalmente deixada no buffer de disco.

A stream associa o arquivo a uma estrutura do tipo FILE. Esta estrutura é definida no arquivo de cabeçalho stdio.h

11.2 Funções utilizadas para manipulação de arquivos

As principais funções para manipulação de arquivos são:

FUNÇÃO	FINALIDADE
fopen( )	Abrir um arquivo
fclose( )	Fechar um arquivo

putc( )	Escrever um caracter em um arquivo
fputc( )	Idem putc( )
getc( )	Ler um caracter de um arquivo
fgetc( )	Idem getc( )
fseek( )	Posicionar o ponteiro de arquivo num byte específico
fprintf( )	É para o arquivo o que **printf** é para o console
fscanf( )	É para o arquivo o que **scanf** é para o console
feof( )	Devolve **verdadeiro** se o fim do arquivo foi atingido
ferror( )	Devolve **verdadeiro** se ocorreu um erro
rewind( )	Posicionar o ponteiro de arquivo no início deste
remove( )	Apagar um arquivo
fflush( )	Descarregar um arquivo

Todas estas funções estão no arquivo de cabeçalho stdio.h.

Este arquivo de cabeçalho define três tipos: **size_t**, **fpos_t** e **FILE**. Os dois primeiros são o mesmo que unsigned e o terceiro é discutido mais abaixo.

Este arquivo de cabeçalho também define várias macros. As importantes para a manipulação de arquivos são: **NULL**, **EOF**, **FOPEN_MAX**, **SEEK_SET**, **SEEK_CUR** e **SEEK_END**.

NULL define um ponteiro nulo.

EOF geralmente é definida como -1 e devolve este valor quando uma função de entrada tenta ler além do final do arquivo.

FOPEN_MAX define um valor inteiro que determina o número de arquivos que podem ser abertos ao mesmo tempo.

SEEK_SET, **SEEK_CUR** e **SEEK_END** são usadas com a função fssek( ) para o acesso aleatório a um arquivo.

11.3 O ponteiro de arquivo

Basicamente um ponteiro de arquivo identifica um arquivo específico e é usado pela stream para direcionar as operações das funções de E/S. Um ponteiro de arquivo é uma variável ponteiro do tipo FILE. Esta variável é um tipo pré-definido pela linguagem C. Normalmente ela é definida no arquivo de cabeçalho stdio.h, mas isso depende do seu compilador. Para ler ou escrever em arquivos seu programa precisa usar os poteiros de arquivo. Para declarar uma variável como ponteiro de arquivo use a seguinte sintaxe:

FILE *arquivo;

11.4 Abrindo um arquivo

Apesar do sistema de E/S de C considerar arquivo como qualquer dispositivo, para os conceitos apresentados daqui pra frente, consideraremos arquivo como um arquivo em disco.

Para abrir uma stream e associá-la a um arquivo você usa a função fopen( ) ,cuja sintaxe é:

fopen(ARQUIVO,MODO)

Onde ARQUIVO é um ponteiro para uma string que representa o nome do arquivo. Na prática é o nome do arquivo propriamente dito e pode ser um PATH, ou seja, algo como "C:\docs\arquivo.txt", no windows; ou algo como "/home/samu/arquivo.txt" no linux. MODO é uma string que representa como o arquivo será aberto de acordo com a tabela abaixo:

MODO	COMO O ARQUIVO SERÁ ABERTO
r	Abre um arquivo texto para leitura.
w	Abre um arquivo texto para escrita. Se um arquivo com o mesmo nome existir, será sobrescrito.
a	Abre um arquivo texto para anexação. Se o arquivo não existir, será criado.
rb	Abre um arquivo binário para leitura.
wb	Abre um arquivo binário para escrita. Se um arquivo com o mesmo nome existir, será sobrescrito.
ab	Abre um arquivo binário para anexação. Se o arquivo não existir, será criado.
r+ w+ a+	Abre um arquivo texto para leitura/escrita. Se o arquivo não existir, será criado.
r+b w+b	Abre um arquivo binário para leitura/escrita. Se o arquivo não existir, será criado.

a+b rb+ wb+ ab+	

É importante lembrar que em muitas implementações, no modo texto, a sequência de caracteres **retorno de carro/alimentação de linha** são traduzidas para **nova linha** na entrada. Na saída ocorre o inverso: caracteres de **nova linha** são convertidos em **retorno de carro/alimentação de linha**. Em arquivos binários não ocorre nenhuma tradução.

Caso tudo corra bem, a função fopen devolve um ponteiro de arquivo, caso ocorra algum problema ela devolve NULL.

Para abrir um arquivo texto chamado "teste" para escrita você poderia escrever assim:

```
FILE *arquivo;


arquivo = fopen("teste","w");
```

Porém, é recomendável sempre testar se o arquivo foi aberto sem problemas. Assim, sempre que for abrir um arquivo, você deve usar um código parecido com este:

```
FILE *arquivo;


if((arquivo = fopen("teste","w")) == NULL)
  {
    printf("Erro ao abrir arquivo!!!\n");
    exit(1);
  }
```

Este tipo de teste detectará algum problema tipo disco cheio ou protegido contra gravação antes que seu programa tente gravar nele.

O número máximo de arquivos que pode ser aberto ao mesmo tempo é definido pela macro FOPEN_MAX, normalmente definida em stdio.h. Confira se é o caso do seu compilador.

11.5 Fechando um arquivo

Para fechar uma stream você deve usar a função fclose( ).

Ela escreve qualquer dado que ainda permanece no buffer de disco no arquivo e o fecha em nível de sistema operacional. Uma falha ao fechar uma stream pode provocar problemas tipo perda de dados, arquivos destruídos e erros intermitentes em seu programa. fclose também libera o bloco de controle de arquivo associado à stream deixando-o disponível para reutilização. Como, normalmente, há um limite do sistema operacional para o número de arquivos abertos ao mesmo tempo, você deve fechar um arquivo antes de abrir outro.

A sintaxe de fclose é:

```
fclose(ARQUIVO);
```

Onde ARQUIVO é o ponteiro de arquivo devolvido por fopen quando esta abriu o arquivo. Caso o fechamento do arquivo ocorra sem problemas, fclose retorna zero. Qualquer outro valor indica erro. Erros possíveis são floppy drive sem disquete ou disco cheio.

11.6 Escrevendo e lendo caracteres

Para escrever um caracter num arquivo aberto você pode usar duas funções: putc( ) ou fputc( ). Elas são identicas. Existem as duas para preservar a compatibilidade com versões mais antigas do C. É lógico que para escrever num arquivo este deve ter sido aberto num modo que permita a escrita.

Veremos putc( ). Sua sintaxe é:

putc(CARACTER,ARQUIVO);

Onde CARACTER é o caracter a ser escrito no arquivo e ARQUIVO é um ponteiro de arquivo.

Se ocorrer tudo bem, a função retorna o caracter escrito, caso contrário ela retorna EOF.

Para ler um caracter temos também duas funções: getc( ) e fgetc( ). Existem duas também pelo motivo da compatibilidade com versões mais antigas da linguagem C. O arquivo deve ter sido aberto num modo que permita a leitura.

Veremos getc( ). Sua sintaxe é:

getc(ARQUIVO);

Onde ARQUIVO é um ponteiro de arquivo.

Quando o final do arquivo é alcançado a função devolve EOF.

Para ler o conteúdo de um arquivo você poderia usar um trecho de código parecido com:

```
do
 {
   caracter = getc(arquivo);
 }
while(caracter != EOF);
```

Sendo caracter uma variável char e arquivo uma variável ponteiro para uma estrutura FILE.

É importante observar que getc também devolve EOF caso ocorra algum erro.

11.7 Programa que lê e exibe o conteúdo de um arquivo texto

```
/* lendo e exibindo o conteúdo de um arquivo texto */

#include <stdio.h>

int main(int argc, char *argv[])
 {
   char caracter;
   FILE *arquivo;
```

```
   if(argc < 2)
    {
      printf("\nErro: Digite o nome do arquivo !!!\n\n");
      exit(1);
    }

   printf("\n%s\n\n",argv[1]);

   if((arquivo = fopen(argv[1],"r")) == NULL)
    {
      printf("Erro ao abrir arquivo!!!\n\n");
      exit(1);
    };

   do
    {
      caracter = getc(arquivo);
      putchar(caracter);
    }
   while(caracter != EOF);

   printf("\n\n");

   fclose(arquivo);

   return(0);
 }
```

Observe que este código exibe também o caracter EOF. Um bom exercício é reescrever este código de modo que este caracter não seja exibido.

11.8 Programa que escreve caracteres num arquivo

```
/* escrevendo caracteres num arquivo */

#include <stdio.h>

int main(int argc, char *argv[])
 {
   FILE *arquivo;
```

```
  char caracter;

  if(argc < 2)
   {
     printf("\nErro: Digite o nome do arquivo !!!\n\n");
     exit(1);
   }

  if((arquivo = fopen(argv[1],"w")) == NULL)
   {
     printf("Erro ao abrir arquivo!!!\n\n");
     exit(1);
   }

  do
   {
     caracter = getchar();
     putc(caracter,arquivo);
   }
  while(caracter != '$');

  fclose(arquivo);

  printf("\nGravado com sucesso em %s\n\n",argv[1]);

  return(0);
 }
```

11.9 Verificando o final de um arquivo binário

Quando manipulando dados binários um valor inteiro igual a EOF pode ser lido por engano. Isso poderia fazer com que fosse indicado o fim de arquivo antes deste ter chegado. Para resolver este problema C inclui a função feof( ) que determina quando o final de um arquivo foi atingido. Ela tem a seguinte sintaxe:

```
feof(ARQUIVO);
```

onde ARQUIVO é um ponteiro de arquivo. Esta função faz parte de stdio.h e devolve verdadeiro caso o final de arquivo seja atingido; caso contrário ela devolve 0.

Para ler um arquivo binário você poderia usar o seguinte trecho de código:

```
while(!feof(arquivo))
  caracter = getc(arquivo);
```

11.10 Programa que copia arquivo

```
/* programa que copia arquivo */

#include <stdio.h>

int main(int argc, char *argv[])
 {
   FILE *original,*copia;
   char caracter;

   if(argc < 3)
    {
      printf("\nSintaxe correta:\n\n");
      printf("copiar ARQUIVO_ORIGEM ARQUIVO_DESTINO\n\n");
      exit(1);
    }


   while(argv[1])
    {
       if(*argv[1] != *argv[2])
         break;
       printf("\nO nome do arquivo original não pode ser igual ao da\
cópia.\n\n");
       exit(1);
    };


   if((original = fopen(argv[1],"rb")) == NULL)
    {
      printf("\nErro ao abrir o arquivo original.\n\n");
      exit(1);
    }

   if((copia = fopen(argv[2],"wb")) == NULL)
    {
      printf("\nErro ao abrir o arquivo cópia.\n\n");
      exit(1);
```

```
    }

  while(!feof(original))
    {
      caracter = getc(original);
      if(!feof(original))
        putc(caracter,copia);
    }

  fclose(original);
  fclose(copia);

  printf("\n%s copiado com sucesso com o nome de %s.\n\n",argv[1],argv[2]);

  return(0);
}
```

11.11 Escrevendo e lendo strings

Para escrever e ler strings em um arquivo use as funções fputs( ) e fgets( ), cujos protótipos encontram-se em stdio.h.

fputs( ) escreve uma string na stream especificada, sua sintaxe é:

fputs(STRING,ARQUIVO);

onde ARQUIVO é um ponteiro de arquivo. Caso ocorra algum erro esta função retorna EOF.

fgets( ) lê uma string da stream especificada. Sua sintaxe é:

fgets(STRING,TAMANHO,ARQUIVO);

Ela lê STRING até que um caractere de nova linha seja lido ou que TAMANHO - 1 caracteres tenham sido lidos. Se uma nova linha é lida ela será parte da string. A string resultante terminará em nulo. Caso ocorra tudo bem esta função retornará um ponteiro para STRING, caso contrário, retornará um ponteiro nulo.

11.12 Programa que escreve strings num arquivo

```
/* Programa que escreve strings num arquivo.
 *
 * Para encerrar o programa o usuário deverá inserir uma
 * linha em branco.
 *
 * Como gets() não armazena o caractere de nova linha,
 * é adicionado um antes da string ser gravada no arquivo.
 * Isto é feito para que a string possa ser lida, posteriormente,
 * com fgets() já que esta função lê a string até que seja
```

```
 * encontrado um caracter de nova linha.
 */

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

int main(int argc, char *argv[])
  {
   char string[80];
   FILE *arquivo;

    if(argc < 2)
      {
        printf("\nErro: Digite o nome do arquivo !!!\n\n");
        exit(1);
      }


    if((arquivo = fopen(argv[1],"w")) == NULL)
      {
        printf("Erro ao abrir arquivo!!!\n\n");
        exit(1);
      }

    do
      {
        gets(string);
        strcat(string,"\n");
        fputs(string,arquivo);
      }
    while(*string != '\n');

    fclose(arquivo);

    return(0);
  }
```

11.13 Apontando para o início do arquivo

Para apontar para o início do arquivo use a função rewind( ), cujo protótipo está no arquivo de cabeçalho stdio.h. Sua sintaxe é:

rewind(ARQUIVO);

sendo ARQUIVO um ponteiro de arquivo.

11.14 Verificando se a operação com o arquivo produziu um erro

Para determinar se uma operação com o arquivo produziu um erro use a função ferror( ). Seu protótipo está em stdio.h e sua sintaxe é:

ferror(ARQUIVO);

sendo ARQUIVO um ponteiro de arquivo. Esta função retorna verdadeiro se ocorreu um erro durante a última operação com o arquivo, caso contrário retorna falso. Como cada operação modifica a condição de erro, ela deve ser chamada logo após cada operação realizada com o arquivo.

11.15 Apagando um arquivo

Para apagar uma arquivo use a função remove( ). Ela faz parte de stdio.h e sua sintaxe é:

remove(ARQUIVO);

sendo ARQUIVO um ponteiro de arquivo.

Exemplo:

```
/* Apagando um arquivo */

#include <stdio.h>
#include <ctype.h>

int main(int argc,char *argv[])
 {
   FILE * arquivo;
   char opcao[5];

   if(argc != 2)
    {
      printf("Erro !!! \n");
      printf("Sintaxe correta: apagar ARQUIVO\n");
      exit(1);
    }

   printf("Deseja realmente apagar o arquivo %s (S/N)?",argv[1]);
   gets(opcao);

   if(toupper(*opcao) == 'S')
```

```
    if(remove(argv[1]))
      {
        printf("Erro ao tentar apagar arquivo.\n");
        exit(1);
      }
    else
      printf("Arquivo apagado com sucesso.\n");

  return(0);
}
```

11.16 Renomeando ou Movendo um arquivo

Para renomear um arquivo use a função rename( ). Ela está em stdio.h e sua sintaxe é:

rename(NOME_ANTIGO, NOVO_NOME);

onde NOME_ANTIGO e NOVO_NOME são ponteiros string. Se a função for bem sucedida ela retornará zero, caso contrário, retornará um valor diferente de zero.

Esta mesma função serve para mover um arquivo. Basta incluir a nova localização como parte do NOVO_NOME do arquivo. Observe o exemplo abaixo:

/* renomeando ou movendo um arquivo */

```
#include <stdio.h>

int main(int argc,char *argv[])
 {

   if(argc != 3)
    {
      printf("Erro !!! \n");
      printf("Sintaxe correta: renomear NOME_ANTIGO NOVO_NOME\n");
      exit(1);
    }

   if(rename(argv[1],argv[2]))
     printf("Erro ao renomear arquivo!\n");

   return(0);
 }
```

Este programa renomeia um arquivo. Porém, se você especificar um outro local para o novo nome do arquivo, este, além de ser renomeado, será movido. Crie um arquivo chamado teste em seu diretório de usuário e, após compilar o exemplo acima com o nome de renomear, digite o seguinte comando:

$ renomear teste teste123

liste o conteúdo do diretório e você verá que o arquivo foi renomeado. Agora crie um diretório chamado lixo e digite o comando:

$ renomear teste123 lixo/teste456

dê uma checada e você verá que o arquivo teste123 não existe mais no diretório atual, mas em lixo foi criado o arquivo teste456, mostrando que teste123 foi movido e renomeado.

11.17 Esvaziando uma stream

Para esvaziar o conteúdo de uma stream aberta para saída use a função fflush( ). Sua sintaxe é:

fflush(ARQUIVO);

sendo ARQUIVO um ponteiro de arquivo. Ela escreve o conteudo do buffer para o arquivo passado como argumento. Se for passado um valor nulo, todos os arquivos abertos para saída serão descarregados. Se tudo ocorrer bem fflush retornará zero, indicando sucesso. Caso contrário, devolverá EOF.

11.18 Escrevendo e lendo tipos de dados definidos pelo usuário

Veremos mais a frente que C permite que o usuário crie seus próprios tipos de daos. Este tipos de dados são estruturas compostas de tipos de dados simples e com estas estruturas podemos construir registros.

Para escrever e ler estas estruturas podemos usar as funções fread( ) e fwrite( ). Elas são definidas em stdio.h.

A sintaxe de fread( ) é:

fread(VARIÁVEL,TAMANHO,QUANTIDADE,ARQUIVO);

onde:

VARIÁVEL é o endereço da variável que receberá os dados lidos do arquivo

TAMANHO é o número de bytes a ser lido. Para calcular isso você deve usar o operador sizeof

QUANTIDADE indica quantos itens serão lidos (cada item do tamanho de TAMANHO)

ARQUIVO é um ponteiro para o arquivo aberto anteriormente

A sintaxe para fwrite( ) é idêntica, com a exceção que VARIÁVEL é o endereço da variável com os dados a serem escritos no arquivo.

O ponteiro de arquivo "ARQUIVO" deve ser aberto em modo binário, para podermos ler e escrever qualquer tipo de informação. Também deve ser aberto de acordo com a operação a ser feita (leitura ou escrita).

fread devolve o número de itens lidos. Esse valor poderá ser menor que QUANTIDADE se o final do arquivo for atingido ou ocorrer um erro.

fwrite devolve o número de itens escritos. Esse valor será igual a QUANTIDADE a menos que ocorra um erro.

Abaixo segue um exemplo do uso de fwrite. Vamos escrever alguns dados num arquivo.

```
/* Usando fwrite para escrever dados num arquivo */

#include <stdio.h>

int main()
 {
   FILE *arquivo;
   char nome[5]="samu";
   int idade=34;
   float altura=1.82;

   if((arquivo = fopen("samu.dat","wb")) == NULL)
     {
       printf("Erro ao abrir arquivo!!!\n\n");
       exit(1);
     }

   fwrite(&nome,sizeof(nome),1,arquivo);
   fwrite(&idade,sizeof(idade),1,arquivo);
   fwrite(&altura,sizeof(altura),1,arquivo);

   fclose(arquivo);

   return(0);
 }
```

Agora veremos um exemplo com o uso de fread que lerá o arquivo "samu.dat" criado no exemplo acima:

```
/* Usando fread para ler dados de um arquivo */

#include <stdio.h>

int main()
 {
   FILE *arquivo;
   char nome[5];
   int idade;
```

```
   float altura;

   if((arquivo = fopen("samu.dat","rb")) == NULL)
    {
      printf("Erro ao abrir arquivo!!!\n\n");
      exit(1);
    }

   fread(&nome,sizeof(nome),1,arquivo);
   fread(&idade,sizeof(idade),1,arquivo);
   fread(&altura,sizeof(altura),1,arquivo);

   printf("nome  : %s\n",nome);
   printf("idade : %d\n",idade);
   printf("altura: %.2f\n",altura);

   fclose(arquivo);

   return(0);
 }
```

Embora não tenha sido feito nestes dois exemplos, é interessante, em programas maiores, a análise do retorno de fread e fwrite para ver se não ocorreram erros. Isto pode ser feito com uma instrução deste tipo:

```
if(fread(&nome,sizeof(nome),1,arquivo) != 1)
  printf("Erro de leitura.\n");
```

11.19 Apontando para uma posição específica dentro do arquivo

Você pode apontar para um byte específico dentro do arquivo movimentando o indicador de posição. Isto pode ser feito com o uso da função fseek( ),cuja sintaxe é:

```
fseek(ARQUIVO, NÚMERO_DE_BYTES, ORIGEM);
```

onde ARQUIVO é um ponteiro de arquivo aberto anteriormente, NÚMERO_DE_BYTES é a quantidade de bytes que o indicador de posição será movimentado e ORIGEM é a partir de onde o movimento do indicador de posição iniciará.

ORIGEM deve ser uma das seguintes macros:

SEEK_SET para a origem no início do arquivo

SEEK_CUR para a origem na posição atual do indicador de posição

SEEK_END para a origem no final do arquivo

estas macros estão definidas em stdio.h.

NÚMERO_DE_BYTES deve ser usado em conjunto com sizeof para que possamos acessar os tipos de dados pré-definidos. Por exemplo, digamos que criamos um tipo de dado chamado "ficha", onde armazenamos os dados de um livro como título, autor, editora, etc... e escrevemos vários dados em um arquivo. Se quisermos apontar o indicador de posição para o quinto registro deste arquivo deveremos usar algo como:

```
fseek(arquivo,4*sizeof(struct ficha),SEEK_SET);
```

11.20 Entrada e saída formatadas direcionadas para arquivos

Existem as funções fprintf( ) e fscanf( ) que são semelhantes a printf e a scanf mas que direcionam os dados para arquivos. Abaixo você pode ver a sintaxe destas funções:

```
fprintf(ARQUIVO,"ESPECIFICADOR",VARIÁVEL);


fscanf(ARQUIVO,"ESPECIFICADOR",VARIÁVEL);
```

onde ARQUIVO é um ponteiro de arquivo aberto para onde são direcionados os resultados das funções.

Abaixo segue um exemplo do uso destas funções:

```
/* exemplo do uso de fscanf e fprintf
 * este programa lê uma string do teclado e a escreve num arquivo texto
 * compilado com o nome de adtexto */


#include <stdio.h>


int main()
 {
   FILE *arquivo;
   char string[80];

   if((arquivo = fopen("adtexto.txt","w")) == NULL)
    {
      printf("Erro ao abrir o arquivo!\n");
      exit(1);
    }

   printf("Exemplo do uso de fscanf e fprintf\n");
   printf("Digite uma string\n\n");
   fscanf(stdin,"%s",string); /* lê string do teclado */
   fprintf(arquivo,"%s",string); /* escreve string no arquivo */
```

```
    fclose(arquivo);


    return(0);
  }
```

Embora sejam uma opção atraente para ler e escrever dados em arquivos, estas funções devem ser evitadas pois trabalham com dados ASCII e não binários o que ocasiona uma perda de desempenho no programa. É preferível o uso de fread e fwrite.

11.21 Streams padrão

Quando um programa em linguagem C é iniciado são abertas três streams: **stdin**, **stdout** e **stderr**.

stdin define a entrada padrão do sistema, normalmente o teclado.

stdout define a saída padrão do sistema, normalmente o monitor.

stderr define a saída padrão dos erros, normalmente também é o monitor.

Estas streams são ponteiros de arquivos e podem ser redirecionadas. Assim, nas funções que você utiliza ponteiros de arquivos para entrada e saída de dados você pode muito bem usar estas streams de modo ao seu programa receber dados do teclado e escrever no monitor. Isto foi mostrado no exemplo da seção anterior na linha;

fscanf(stdin,"%s",string); /* lê string do teclado */

onde o programa leu a variável string do teclado através da streams padrão stdin.

Porém esteja consciente que estas streams não são variáveis e não podem receber um valor. Ou seja, você não pode abri-las com fopen.

Quando o programa é encerrado estas streams são fechadas automaticamente, do mesmo jeito que foram criadas, você não deve nunca tentar abrí-las ou fechá-las.

12. Matrizes

12.1 Introdução as matrizes

Uma matriz é uma estrutura de dados que pode armazenar vários valores do mesmo tipo.

A sintaxe para declarar uma matriz é:

TIPO NOME[QUANTIDADE];

onde TIPO é o tipo dos dados que serão armazenados na matriz. Todos os dados colocados na matriz devem ser deste tipo. NOME é o nome a ser dado a matriz. Este nome identificará a matriz no código do programa. E QUANTIDADE é a quantidade máxima de itens a ser armazenados. Exemplos:

int nr_de_livros[50]; /* esta matriz pode armazenar até 50 valores do tipo int */

float nota[30]; /* esta matriz pode armazenar até 30 valores do tipo float */

Os valores armazenados na matriz são chamados de "elementos da matriz". O primeiro elemento da matriz é indexado como item zero e o último é indexado como QUANTIDADE menos 1. Assim, para nossa matriz nota, mostrada no exemplo acima, o primeiro elemento é nota[0] e o último elemento é nota[29].

Você pode inicializar os elementos de uma matriz na sua declaração usando a sintaxe:

int notas[5] = {60,70,35,50,68};

No exemplo acima o elemento zero da matriz notas receberá o valor 60, o elemento 1 receberá o valor 70, e assim por diante. Para melhorar o entendimento observe o código abaixo:

```
#include <stdio.h>

int main()
 {
   int notas[5] = {60,70,35,50,68};

   printf("Analisando os elementos da matriz notas\n");
   printf("O primeiro elemento tem o valor %d\n",notas[0]);
   printf("O segundo elemento tem o valor %d\n",notas[1]);
   printf("O terceiro elemento tem o valor %d\n",notas[2]);
   printf("O quarto elemento tem o valor %d\n",notas[3]);
   printf("O quinto e último elemento tem o valor %d\n",notas[4]);

   return(0);
 }
```

Este código pode ser otimizado usando um laço for e uma variável para manipular os elementos da matriz. Observe abaixo;

```
#include <stdio.h>

int main()
 {
   int notas[5] = {60,70,35,50,68};
   int contador;

   printf("Analisando os elementos da matriz notas\n");
   for(contador = 0;contador < 5;contador++)
     printf("O %do elemento tem o valor %d\n",contador+1,notas[contador]);
   return(0);
 }
```

Uma das matrizes mais comuns utilizadas em C é a matriz de caracteres. As strings manipuladas em C são matrizes de caracteres. Observe o exemplo abaixo para um melhor entendimento:

```
/* visualizando strings como matrizes de caracteres */


#include <stdio.h>


int main()
 {
   char palavra[7] = "matriz";
   int contador;


   printf("Em C strings são matrizes de caracteres e podem ser manipuladas como tal.\n");
   printf("\nA string é %s\n",palavra);
   printf("\nExibindo cada elemento da matriz palavra\n");
   for(contador = 0;contador < 7;contador++)
     printf("%c\n",palavra[contador]);


   return(0);
```

12.2 Passando uma matriz para uma função

Uma função que manipula uma matriz deve receber a matriz e a quantidade de elementos. Logicamente ao chamar a função você deve passar a matriz propriamente dita e seu número de elementos. Não há necessidade de passar o tamanho da matriz. Exemplo:

```
#include <stdio.h>


void exibe(int matriz[],int elementos)
 {
   int contador;


   for(contador = 0;contador < elementos;contador++)
     printf("O %do elemento tem o valor %d\n",contador+1,matriz[contador]);
 }



int main()
 {
   int notas[5] = {60,70,35,50,68};
   int contador;
```

```
    printf("Analisando os elementos da matriz notas\n");

    exibe(notas,5);

    return(0);
  }
```

Observe os colchetes após a variável matriz na declaração da função exibe. Isto indica que a variável é uma matriz.

12.3 Matrizes bidimensionais

Imagine uma matriz bidimensional como uma tabela de linhas e colunas. Por exemplo, a matriz

pesos[3][5]

pode ser imaginada como:

Observe que o primeiro índice ([3]) indica as linhas da matriz e o segundo ([5]) indica as colunas.

Como sabemos que [3] varia de zero a 2 e [5] varia de zero a 4, fica fácil determinar os índices de cada posição da matriz:

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

Visto a posição de cada índice vamos preencher nossa matriz pesos com valores

10	30	45	70	36
86	44	63	82	80
70	61	52	63	74

De tudo que foi exposto acima podemos entender que:

pesos[1][3] = 82

pesos[0][4] = 36

pesos[0][0] = 10

pesos[2][4] = 74

Para preencher nossa matriz com os valores mostrados na tabela acima podemos usar uma declaração como:

```
int pesos[3][5] = {{10,30,45,70,36},
                   {86,44,63,82,80},
                   {70,61,52,63,74}};
```

Podemos manipular os elementos de nossa matriz bidimensional usando duas variáveis e um laço for da mesma maneira que fizemos com as matrizes comuns. Observe o código abaixo:

```
/* manipulando uma matriz bidimensional */
#include <stdio.h>

int main()
 {
   int pesos[3][5] = {{10,30,45,70,36},
                      {86,44,63,82,80},
                      {70,61,52,63,74}};

   int linha,coluna;

   for(linha = 0;linha < 3;linha++)
     for(coluna = 0;coluna < 5; coluna++)
       printf("elemento[%d][%d] = %d\n",linha,coluna,pesos[linha][coluna]);

   return(0);
 }
```

12.4 Passando uma matriz bidimensional para uma função

Uma função que manipula uma matriz bidimensional deve receber a matriz e o número de linhas desta matriz. O número de colunas da matriz também deve estar especificado nesta declaração. Ao chamar a função, deve-se passar a matriz e o número de linhas. Exemplo:

```
/* manipulando uma matriz bidimensional */
#include <stdio.h>

void exibe(int matriz[][5], int linhas)
 {
   int linha,coluna;
```

```
    for(linha = 0;linha < 3;linha++)
      for(coluna = 0;coluna < 5; coluna++)
        printf("elemento[%d][%d] = %d\n",linha,coluna,matriz[linha][coluna]);
 }


int main()
 {
    int pesos[3][5] = {{10,30,45,70,36},
                       {86,44,63,82,80},
                       {70,61,52,63,74}};

    exibe(pesos,3);

    return(0);
 }
```

12.5 Pesquisa sequencial

Para procurar um valor específico numa matriz você pode usar a pesquisa sequencial. A pesquisa sequencial inicia no primeiro elemento da matriz e vai até o último procurando o valor desejado. Observe abaixo o código de uma pesquisa sequencial numa matriz:

```
/* exemplo de uma pesquisa sequencial numa matriz */


#include <stdio.h>


int main()
 {
   int pesos[5]={65,77,84,80,69};
   int ok=0,contador,valor;

   printf("\nmatriz pesos\n\n");
   for(contador = 0;contador < 5;contador++)
     printf("pesos[%d] = %d\n",contador,pesos[contador]);

   printf("\nEntre com o valor a ser pesquisado :");
   scanf("%d",&valor);

   /* realizando a pesquisa sequencial */
```

```
  contador=0;
  while((contador < 5) && (!ok))
    if(pesos[contador] == valor)
      ok = 1;
    else
      contador++;

  if(contador < 5)
    printf("O valor %d está no elemento pesos[%d]\n",valor,contador);
  else
    printf("A matriz pesos não possui o valor %d\n",valor);

  return(0);
}
```

Este tipo de pesquisa pode ser demorado quando a matriz possui muitos elementos. Nestes casos use a pesquisa binária que será vista mais a frente.

12.6 Ordenando os elementos de uma matriz pelo método da bolha

Este método, apesar de simples, consome mais tempo do processador, então só deve ser usado para matrizes com poucos elementos (em torno de 30). Neste método os elementos da matriz vão sendo percorridos e os pares adjacentes são comparados e ordenados. Isto é feito até que toda a matriz esteja ordenada.

Para explicar melhor vamos a um exemplo prático. Imagine a seguinte matriz:

nota[0] = 67

nota[1] = 55

nota[2] = 86

nota[3] = 79

nota[4] = 68

Ordenando esta matriz em ordem crescente pelo método da bolha, nosso programa iniciaria compararando o elemento nota[0], que é 67, com o elemento nota[1], que é 55. Como nota[0] é maior que nota[1] os elementos seriam trocados. e nossa matriz ficaria assim:

nota[0] = 55

nota[1] = 67

nota[2] = 86

nota[3] = 79

nota[4] = 68

Agora nosso programa compararia nota[1] com nota[2] e como os valores estão ordenados não haveria troca.

Continuando, compararia nota[2] com nota[3] e haveria troca ficando nossa matriz assim:

nota[0] = 55

nota[1] = 67

nota[2] = 79

nota[3] = 86

nota[4] = 68

Agora compararia nota[3] com nota[4] e novamente haveria troca:

nota[0] = 55

nota[1] = 67

nota[2] = 79

nota[3] = 68

nota[4] = 86

Agora, nosso programa iniciaria novo ciclo de comparações comparando novamente nota[0] com nota[1], nota[1] com nota[2], etc .... até que todos os elementos da matriz estivessem na ordem crescente.

Abaixo segue um exercício que ordena uma matriz de cinco valores inteiros, em ordem crescente, pelo método da bolha:

```
/* ordenando uma matriz pelo método da bolha *
 *
 * neste método a matriz é percorrida e os pares
 * adjacentes são comparados e ordenados. Isto é
 * feito até que toda a matriz esteja ordenada */

#include <stdio.h>

int main()
 {
   int vetor[5],contador,ordenados,auxiliar;

   printf("Ordenando uma matriz utilizando o método da bolha.\n");

   /* entrada de dados */
   for(contador=0;contador < 5; contador++)
    {
      printf("Entre com o %do valor :",(contador+1));
      scanf("%d",&vetor[contador]);
    }

   /* ordenação */
```

```
   ordenados = 0; /* indica que os elementos adjacentes não estão ordenados
*/
   while(ordenados == 0)
    {
      ordenados = 1; /* considera todos os elementos ordenados corretamente
*/
      for(contador=0;contador < 4;contador++)
       {
       /* se os elementos adjacentes não estiverem ordenados executa a troca
*/
         if(vetor[contador] > vetor[(contador + 1)])
          {
            auxiliar = vetor[contador];
            vetor[contador] = vetor[(contador + 1)];
            vetor[(contador + 1)] = auxiliar;
            ordenados = 0; /* força outra passagem no laço while */
          }
       }
    }

   /* imprimindo os valores ordenados */
   printf("\n");
   for(contador=0;contador < 5;contador++)
     printf("%d ",vetor[contador]);
   printf("\n");

   return(0);
 }
```

12.7 Ordenando os elementos de uma matriz pelo método da seleção

Este método também só deve ser usado com matrizes pequenas. Neste método o programa procura o menor valor entre todos os elementos da matriz e troca-o pelo primeiro elemento; depois procura o menor entre o segundo e o último elemento da matriz e troca-o pelo segundo elemento; depois procura o menor valor entre o terceiro e o último elemento da matriz e troca-o pelo terceiro, e assim por diante.

Vamos ver um exemplo prático. Dada a matriz:

nota[0] = 67

nota[1] = 55

nota[2] = 86

nota[3] = 79

nota[4] = 68

Vamos ordená-la em ordem crescente pelo método da seleção:

Primeiro nosso programa teria que determinar o menor valor da matriz, que é 55 e trocá-lo pelo primeiro elemento da matriz (nota [0]). Nossa matriz ficaria assim:

nota[0] = 55

nota[1] = 67

nota[2] = 86

nota[3] = 79

nota[4] = 68

Agora seria determinado o menor valor entre o segundo (nota [1]) e o último elemento (nota [4]), que é 67, e este seria trocado com o segundo elemento (nota [1]), ficando a matriz da seguinte maneira:

nota[0] = 55

nota[1] = 67

nota[2] = 86

nota[3] = 79

nota[4] = 68

Agora seria determinado o menor valor entre o terceiro (nota [2]) e o último elemento (nota [4]), que é 68, e este seria trocado com o terceiro elemento (nota [2]). Agora a matriz está assim:

nota[0] = 55

nota[1] = 67

nota[2] = 68

nota[3] = 79

nota[4] = 86

Agora seria determinado o menor valor entre nota [3] e nota [4](no caso 79) e este valor trocado com o quarto elemento da matriz. E nossa matriz estaria ordenada:

nota[0] = 55

nota[1] = 67

nota[2] = 68

nota[3] = 79

nota[4] = 86

Abaixo segue um exercício que ordena uma matriz de cinco valores inteiros, em ordem crescente, pelo método da seleção:

```
/* ordenando uma matriz pelo método da seleção *
 *
 * neste método a ordenação é feita da seguinte maneira:
 *   - identifica-se o menor valor da matriz
```

```
 *   - troca-se este com o primeiro elemento
 *   - identifica-se o menor valor entre o segundo e o último elemento
 *   - troca-se este com o segundo elemento
 *   - identifica-se o menor valor entre o terceiro e o último elemento
 *   - troca-se este com o terceiro elemento
 *   - identifica-se o menor valor entre o quarto e o último elemento
 *   - troca-se este com o quarto elemento
 *
 * e assim por diante, até que toda a matriz esteja ordenada */

#include <stdio.h>

int main()
 {
   int vetor[5],contador1,contador2,indice_do_menor, menor;

   printf("Ordenando uma matriz utilizando o método da seleção.\n");

   /* entrada de dados */
   for(contador1 = 0;contador1 < 5; contador1++)
    {
      printf("Entre com o %do valor :",(contador1+1));
      scanf("%d",&vetor[contador1]);
    }

   /* ordenação */
   for(contador1 = 0;contador1 < 4;contador1++)
    {
      indice_do_menor = contador1;
      menor = vetor[contador1];

      /* verificando qual o menor */
      for(contador2=(contador1 + 1);contador2 < 5;contador2++)
        if(vetor[contador2] < menor)
          {
            indice_do_menor = contador2;
            /* executando a troca de valores */
            menor = vetor[indice_do_menor];
```

```
            vetor[indice_do_menor] = vetor[contador1];
            vetor[contador1] = menor;
          }
      }

    /* imprimindo os valores ordenados */
    printf("\n");
    for(contador1 = 0;contador1 < 5;contador1++)
      printf("%d ",vetor[contador1]);
    printf("\n");

    return(0);
  }
```

12.8 Ordenando os elementos de uma matriz pelo método da inserção

Neste método, também usado com matrizes pequenas, o programa inicialmente ordena os dois primeiros elementos da matriz. Depois insere o terceiro elemento na posição ordenada em relação aos dois primeiros. Depois insere o quarto elemento em sua posição ordenada em relação aos três já posicionados, e assim por diante até que toda a matriz esteja ordenada.

Vamos a um exemplo prático. Dada nossa matriz:

nota[0] = 67

nota[1] = 55

nota[2] = 86

nota[3] = 79

nota[4] = 68

Vamos ordená-la em ordem crescente pelo método da inserção:

Primeiro nosso programa ordenaria os dois primeiros elementos da matriz:

nota[0] = 55

nota[1] = 67

Depois o programa pegaria o terceiro elemento da matriz e o introduziria na posição ordenada em relação aos dois primeiros:

nota[0] = 55

nota[1] = 67

nota[2] = 86

Depois o programa pegaria o quarto elemento da matriz e o introduziria na posição ordenada em relação aos três primeiros:

nota[0] = 55

nota[1] = 67

nota[2] = 79

nota[3] = 86

Depois o programa pegaria o quinto elemento da matriz e o introduziria na posição ordenada em relação aos quatro primeiros:

nota[0] = 55

nota[1] = 67

nota[2] = 68

nota[3] = 79

nota[4] = 86

E nossa matriz estaria ordenada.

Abaixo segue um exercício que ordena uma matriz de cinco valores inteiros, em ordem crescente, pelo método da inserção:

```
/* ordenando uma matriz pelo método da inserção *
 *
 * neste método a ordenação é feita da seguinte maneira:
 *  - os dois primeiros elementos da matriz são ordenados
 *  - o terceiro elemento é introduzido em sua posição ordenada em relação aos dois primeiros
 *  - o quarto elemento é introduzido em sua posição ordenada em relação aos três primeiros
 *  - o quinto elemento é introduzido em sua posição ordenada em relação aos quatro primeiros
 *
 *  e assim por diante até que toda a matriz esteja ordenada.*/


#include <stdio.h>


int main()
 {
   int vetor_original[5], vetor_ordenado[5];
   int contador,contador2,contador3;
   int ok = 0;


   printf("Ordenando uma matriz utilizando o método da inserção.\n");


   /* entrada de dados */
   for(contador=0;contador < 5; contador++)
    {
      printf("Entre com o %do valor :",(contador+1));
      scanf("%d",&vetor_original[contador]);
    }
```

```
/* ordenação */

/* ordenando os dois primeiros elementos */
if(vetor_original[0] < vetor_original[1])
  {
    vetor_ordenado[0] = vetor_original[0];
    vetor_ordenado[1] = vetor_original[1];
  }
else
  {
    vetor_ordenado[0] = vetor_original[1];
    vetor_ordenado[1] = vetor_original[0];
  }

/* ordenando os outros elementos */
for(contador = 2;contador < 5; contador++)
  {
     /* introduzindo o elemento vetor_original[contador] no vetor_ordenado */

     ok = 1; /* considera vetor_ordenado não ordenado */

     /* verificando se algum dos elementos de vetor_ordenado é maior que
      * vetor_original[contador] */
     for(contador2 = 0;contador2 < contador;contador2++)
     {
        if(vetor_ordenado[contador2] > vetor_original[contador])
          {
             /* jogando os elementos de vetor_ordenado para frente */
             for(contador3 = contador;contador3 > contador2;contador3--)
               vetor_ordenado[contador3] = vetor_ordenado[contador3 - 1];
             /* inserindo vetor_original[contador] em sua posição ordenada */
             vetor_ordenado[contador2] = vetor_original[contador];
             ok =0; /* considerando vetor_ordenado ordenado */
             break;
          }
      }
```

```
        /* se nenhum elemento de vetor_ordenado é maior que
vetor_original[contador] ... */
        if(ok == 1)
          vetor_ordenado[contador] = vetor_original[contador];
     }

   /* imprimindo os valores ordenados */
   printf("\n");
   for(contador = 0;contador < 5;contador++)
     printf("%d ",vetor_ordenado[contador]);
   printf("\n");

   return(0);

 }
```

12.9 Ordenando os elementos de uma matriz pelo método shell

Este método de ordenação oferece uma performance bem melhor que os anteriormente mostrados. Nele compara-se elementos separados por uma distância específica ordenando-os. Esta distância então é dividida por dois e o processo continua até que a matriz esteja ordenada.

Vamos ao nosso exemplo prático com a matriz:

nota[0] = 67

nota[1] = 55

nota[2] = 86

nota[3] = 79

nota[4] = 68

nota[5] = 99

nota[6] = 0

nota[7] = 34

nota[8] = 18

nota[9] = 27

Nossa matriz tem dez elementos. Vamos começar nossa ordenação ordenando os elementos separados de cinco posições. Então compararemos nota[0] e nota[5]. Como estes elementos estão ordenados, nosso programa nada faria.

Agora serão comparados nota[1] e nota[6]. Como estão fora de ordem, serão ordenados e nossa matriz ficará assim:

nota[0] = 67

nota[1] = 0

nota[2] = 86

nota[3] = 79

nota[4] = 68

nota[5] = 99

nota[6] = 55

nota[7] = 34

nota[8] = 18

nota[9] = 27

Agora nota[2] e nota[7]. Como estão desordenados, haverá ordenação e a matriz ficará assim:

nota[0] = 67

nota[1] = 0

nota[2] = 34

nota[3] = 79

nota[4] = 68

nota[5] = 99

nota[6] = 55

nota[7] = 86

nota[8] = 18

nota[9] = 27

Agora nota[3] e nota[8]:

nota[0] = 67

nota[1] = 0

nota[2] = 34

nota[3] = 18

nota[4] = 68

nota[5] = 99

nota[6] = 55

nota[7] = 86

nota[8] = 79

nota[9] = 27

E nota[4] com nota[9]:

nota[0] = 67

nota[1] = 0

nota[2] = 34

nota[3] = 18

nota[4] = 27

nota[5] = 99

nota[6] = 55

nota[7] = 86

nota[8] = 79

nota[9] = 68

Terminado este ciclo de comparações a nossa distância será dividida por dois, sendo agora igual a três, já que devemos usar números inteiros para definir esta distância. Agora vamos a outro ciclo de comparações, só que desta vez com os elementos separados por três posições.

Iniciando com a comparação de nota[0] e nota[3], haveria ordenação e nossa matriz agora ficará assim:

nota[0] = 18

nota[1] = 0

nota[2] = 34

nota[3] = 67

nota[4] = 27

nota[5] = 99

nota[6] = 55

nota[7] = 86

nota[8] = 79

nota[9] = 68

Agora nota[1] com nota[4], como estão ordenados não há nenhuma mudança.

Agora nota[2] com nota[5], também sem mudanças.

nota[3] com nota[6]. Estão desordenados. Nossa matriz agora ficará assim:

nota[0] = 18

nota[1] = 0

nota[2] = 34

nota[3] = 55

nota[4] = 27

nota[5] = 99

nota[6] = 67

nota[7] = 86

nota[8] = 79

nota[9] = 68

Agora nota[4] com nota[7]. Como os elementos estão ordenados, não há mudanças.

nota[5] com nota[8] estão desordenados, então haverá reordenação:

nota[0] = 18

nota[1] = 0

nota[2] = 34

nota[3] = 55

nota[4] = 27

nota[5] = 79

nota[6] = 67

nota[7] = 86

nota[8] = 99

nota[9] = 68

E terminando este ciclo a comparação de nota[6] com nota[9] não alterará a matriz pois os elementos estão ordenados.

Agora a distância é novamente dividia por dois e nossa nova distância será dois, pois não podemos usar a distância 1,5.

Abaixo temos as comparações deste novo ciclo;

nota[0] com nota[2] => estão ordenados, matriz inalterada.

nota[1] com nota[3] => estão ordenados, matriz inalterada.

nota[2] com nota[4] => estão desordenados, a matriz fica assim:

nota[0] = 18

nota[1] = 0

nota[2] = 27

nota[3] = 55

nota[4] = 34

nota[5] = 79

nota[6] = 67

nota[7] = 86

nota[8] = 99

nota[9] = 68

nota[3] com nota[5] => estão ordenados, matriz inalterada.

nota[4] com nota[6] => estão ordenados, matriz inalterada.

nota[5] com nota[7] => estão ordenados, matriz inalterada.

nota[6] com nota[8] => estão ordenados, matriz inalterada.

nota[7] com nota[9] => estão desordenados, a matriz fica assim:

nota[0] = 18

nota[1] = 0

nota[2] = 27

nota[3] = 55

nota[4] = 34

nota[5] = 79

nota[6] = 67

nota[7] = 68

nota[8] = 99

nota[9] = 86

E agora o ciclo final. Dividindo nossa distância atual (2) por dois teremos a comparação dos elementos separados por uma posição.E ao final deste ciclo nossa matriz estaria ordenada.

O único detalhe a ser observado é evitar sequências de distâncias que são potência de dois, pois elas reduzem a eficiência deste algoritmo.Por exemplo: a sequência "8,4,2,1" não é interessante,seria melhor "7,3,2,1".

Abaixo segue uma codificação deste exemplo:

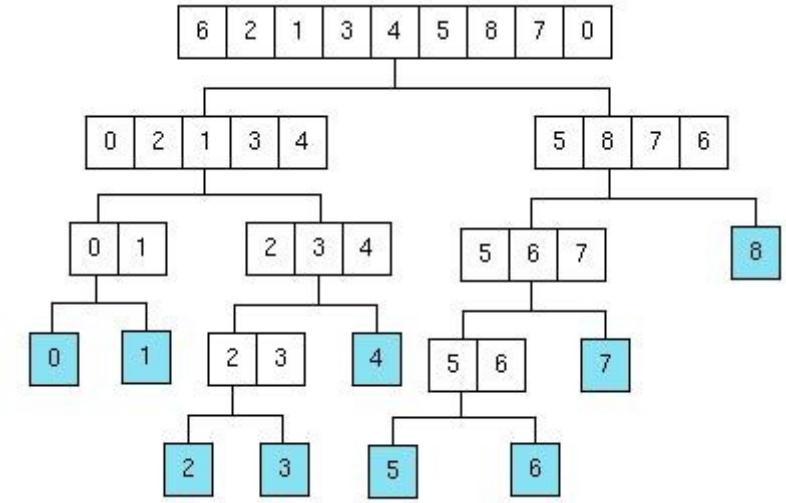
```
/* ordenando uma matriz pelo método shell *
 *
 * neste método a ordenação é feita da seguinte maneira:
 *   - compara-se os elementos separados por uma distância específica
 *   - divide-se esta distância por dois
 *   - compara-se os elementos separados pela nova distância
 *   - divide-se a distância por dois novamente
 * e assim por diante, até que toda a matriz esteja ordenada */

#include <stdio.h>

int main()
 {
   int QDE = 10;
   int vetor[QDE],contador1,contador2,distancia,auxiliar;

   printf("Ordenando uma matriz utilizando o método shell.\n");

   /* entrada de dados */
   for(contador1 = 0;contador1 < QDE; contador1++)
    {
      printf("Entre com o %do valor :",(contador1+1));
      scanf("%d",&vetor[contador1]);
    }

   /* ordenação */
   distancia = QDE;
   do
    {
      distancia = (distancia + 1) / 2;
      for(contador2 = 0;contador2 < (QDE - distancia); contador2++)
       {
```

```
          if(vetor[contador2] > vetor[contador2 + distancia])
            {
              auxiliar = vetor[contador2 + distancia];
              vetor[contador2 + distancia] = vetor[contador2];
              vetor[contador2] = auxiliar;
            }
        }
      }
    while (distancia > 1);

    /* imprimindo os valores ordenados */
    printf("\n");
    for(contador1 = 0;contador1 < QDE;contador1++)
      printf("%d ",vetor[contador1]);
    printf("\n");

    return(0);
  }
```

12.10 Ordenando os elementos de uma matriz pelo método quick sort

Para matrizes com muitos elementos este é o método de ordenação mais rápido. Neste tipo de ordenação o programa considera sua matriz uma lista de valores. Inicialmente é selecionado o valor que está posicionado no meio da lista que chamaremos de elemento central. Depois a matriz é dividida e ordenada em duas listas menores separando numa os elementos cujo valor é maior que o valor do elemento central e e na outra os elementos cujo valor é menor que o valor do elementro central. A partir daí o mesmo processo é feito com cada uma das listas recursivamente. Isso continua até que a matriz esteja toda ordenada.

Considere a matriz formada pelos valores inteiros: 6, 2, 1, 3, 4, 5, 8, 7 e 0. A figura abaixo mostra como poderíamos classificar esta matriz em ordem crescente usando o quick sort.



Abaixo segue o exemplo de um código que ordena em ordem crescente, pelo método quick sort, uma matriz de valores inteiros:

/* ordenando uma matriz pelo método quick sort */

```
#include <stdio.h>

void quick_sort(int matriz[], int primeiro, int ultimo)
 {
   int temp, baixo,alto,separador;
   baixo = primeiro;
   alto = ultimo;
   separador = matriz[(primeiro + ultimo) / 2];
   do
    {
      while(matriz[baixo] < separador)
          baixo++;
      while(matriz[alto] > separador)
        alto--;
    if(baixo <= alto)
     {
          temp =  matriz[baixo];
          matriz[baixo++] = matriz[alto];
          matriz[alto--] = temp;
     }
    }
  while(baixo <= alto);
  if(primeiro < alto)
```

```
    quick_sort(matriz,primeiro,alto);
  if(baixo < ultimo)
    quick_sort(matriz,baixo,ultimo);
 }

int main()
 {
   int QDE = 10;
   int vetor[QDE],contador;

   printf("Ordenando uma matriz utilizando o método quick sort.\n");

   /* entrada de dados */
   for(contador = 0;contador < QDE; contador++)
     {
       printf("Entre com o %do valor :",(contador+1));
       scanf("%d",&vetor[contador]);
     }

   quick_sort(vetor,0,(QDE-1));

  /* imprimindo os valores ordenados */
   printf("\n");
   for(contador = 0;contador < QDE;contador++)
     printf("%d ",vetor[contador]);
   printf("\n");

   return(0);
 }
```

13. Ponteiros

13.1 Exibindo o endereço de memória de uma variável

Para exibir o endereço de memória de uma variável use o operador & antes da variável. Lembre-se que o **valor da variável** é uma coisa e o **endereço de memória** onde este valor está armazenado é outra. O código abaixo esclarecerá melhor estes conceitos:

```
/* exibindo o endereço de memória de variáveis */
```

```
#include <stdio.h>

int main()
 {
   char letra = 's';
   int idade = 35;
   char nome[10] = "samuel";
   float peso = 87.8;
   float altura = 1.82;

   printf("Exibindo o endereço de memória de variáveis\n\n");
   printf("O valor da variável letra é %c e seu endereço é %x\n",letra,&letra);
   printf("O valor da variável idade é %d e seu endereço é %x\n",idade,&idade);
   printf("O valor da variável nome é %s e seu endereço é %x\n",nome,nome);
   printf("O valor da variável peso é %2.1f e seu endereço é
%x\n",peso,&peso);
   printf("O valor da variável altura é %1.2f e seu endereço é
%x\n",altura,&altura);

 }
```

Ao ser executado, o código acima deverá exibir algo parecido com:

Exibindo o endereço de memória de variáveis

O valor da variável letra é s e seu endereço é bffff8cb

O valor da variável idade é 35 e seu endereço é bffff8c4

O valor da variável nome é samuel e seu endereço é bffff8b8

O valor da variável peso é 87.8 e seu endereço é bffff8b4

O valor da variável altura é 1.82 e seu endereço é bffff8b0

Observe na codificação que a variável nome não precisou do operador de endereço & pois matrizes já são tratadas como ponteiros pela linguagem C. Assim, quando o programa passa uma matriz para uma função, o compilador passa o endereço inicial da matriz.

13.2 Definição de ponteiros

Depois de entendida a diferença entre o valor de uma variável e seu endereço de memória fica fácil definir ponteiros.

Ponteiro é uma variável que armazena um endereço de memória.

OBSERVAÇÃO: Cabe aqui ressaltar que a linguagem C trata as matrizes como ponteiros, assim quando seu programa passa uma matriz para uma função, o

compilador na verdade passa o endereço do primeiro elemento da matriz. Observe o exemplo abaixo:

/* verificando como C trata as matrizes como ponteiros */

```
#include <stdio.h>

int main()
 {
   int idade[5];
   float peso[5];
   char nome[10] = "samuel";

   printf("O endereço do primeiro elemento da matriz idade é %x\n",idade);
   printf("O endereço do primeiro elemento da matriz peso é %x\n",peso);
   printf("O endereço do primeiro elemento da matriz nome é %x\n",nome);

   return(0);
 }
```

13.3 Declarando uma variável ponteiro

A sintaxe para a declaração de uma variável ponteiro é:

TIPO *NOME

onde TIPO é um tipo de dados válido e NOME é o nome da variável. Exemplo:

int *idade;

13.4 Atribuindo valor a uma variável ponteiro

Como o ponteiro é uma variável que armazena um endereço de memória, você deverá atribuir a uma variável ponteiro um endereço de memória. Exemplo:

/* atribuindo valor a uma variável ponteiro */

```
#include <stdio.h>

int main()
 {
   int idade = 35;
   int *ptr_idade;

   ptr_idade = &idade; /* foi atribuído o endereço da variável
                          idade a variável ponteiro ptr_idade.
                          Observe o uso do operador & que devolve
```

```
                    o endereço de memória da variável
                    idade. */

  printf("O valor da variável idade é %d\n",idade);
  printf("O endereço da variável idade é %x\n",&idade);
  printf("O valor da variável ponteiro ptr_idade é %x\n",ptr_idade);

  return(0);
 }
```

13.5 Desreferenciando um ponteiro

Um ponteiro armazena um endereço de memória. Desreferenciar um ponteiro é acessar o valor armazenado neste endereço. Para isso você deve usar o operador de indireção, que é o asterisco * . Exemplo:

```
/* desreferenciando um ponteiro */

#include <stdio.h>

int main()
 {
  int idade = 35;
  int *ptr_idade;

  ptr_idade = &idade; /* foi atribuído o endereço da variável
                    idade a variável ponteiro ptr_idade.
                    Observe o uso do operador & que devolve
                    o endereço de memória da variável
                    idade. */

  printf("O valor da variável idade é %d\n",idade);
  printf("O endereço da variável idade é %x\n",&idade);
  printf("O valor da variável ponteiro ptr_idade é %x\n",ptr_idade);

  printf("O valor apontado por ptr_idade é %d\n",*ptr_idade);
  /* observe, na linha acima, o uso do operador de indireção ( * )
   * para desreferenciar o ponteiro ptr_idade e, assim, exibir
   * o valor armazenado no endereço de memória apontado por ele. */
```

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

13.6 Alterando o valor armazenado no endereço apontado por um ponteiro

```
/* Alterando o valor armazenado no endereço apontado
 * por um ponteiro
 */

#include <stdio.h>

int main()
 {
   int numero = 35;
   int *ptr;

   ptr = &numero; /* atribuindo o endereço de numero a ptr */

   printf("O ponteiro ptr armazena o endereço %x que,\npor sua vez,\
armazena o valor %d\n",ptr,*ptr);

   *ptr = 25; /* alterando o valor armazenado no endereço
              * apontado por ptr. Observe que o ponteiro
                 * deve ser desreferenciado.*/

   printf("\nAgora o ponteiro ptr armazena o endereço %x que,\npor sua vez,\
armazena o valor %d\n",ptr,*ptr);

   return(0);
 }
```

13.7 Ponteiros como parâmetros de função

Quando queremos alterar o valor dos argumentos passados para uma função devemos definir os parâmetros da função como ponteiros. A isso denominamos chamada por referência. Exemplo:

```
/* ponteiros como parâmteros de função */

#include <stdio.h>

/* a função que recebe como argumento o valor
```

```
 * da variável não consegue alterar o valor
 * deste
 */
int valor(int a)
 {
   a = 35; /* alterando o valor do argumento passado */
 }

/* a função que recebe como argumento um ponteiro
 * consegue alterar o valor apontado por este
 */
int ponteiro(int *a)
 {
   *a  = 35; /* alterando o valor do argumento passado */
 }



int main()
 {
   int nr = 26;
   int *ptr_nr;

   printf("O valor inicial de nr é %d\n",nr);

   valor(nr); /* função que recebe o valor. Não consegue alterar este */
   printf("Valor de nr após a chamada da função valor = %d\n",nr);

   ptr_nr = &nr;
   ponteiro(ptr_nr); /* função que recebe ponteiro. Consegue alterar valor
                      * apontado
                      */
   printf("Valor de nr após a chamada da função ponteiro = %d\n",nr);

   return(0);
 }
```

13.8 Aritmética dos ponteiros

Você pode somar e subtrair valores a ponteiros, porém deve estar atento a um detalhe. Os ponteiros são endereços de memória, assim ao somar 1 a um

ponteiro você estará indo para o próximo **endereço de memória** do tipo de dado especificado. Por exemplo, digamos que um ponteiro do tipo char aponta para o endereço 1000. Se você somar 2 a este ponteiro o resultado será um ponteiro apontando para o endereço 1002, pois o tipo char requer um byte de memória para armazenar seus dados. Se este ponteiro fosse do tipo int o resultado seria um ponteiro apontando para o endereço 1008, pois o tipo int, sob o linux, requer quatro bytes para armazenar seus dados. Então, sempre que for somar ou subtrair aos ponteiros você tem que trabalhar com o tamanho do tipo de dado utilizado. Para isso você pode usar o operador sizeof.

Analise o código abaixo para um melhor entendimento deste conceito:

```
/* visualizando como funciona a aritmética de ponteiros */


#include <stdio.h>


int main()
  {
    char letra[5] = {'a','e','i','o','u'};
    int contador, nr[5] = {30,12,67,13,41};

    char *ptr_letra;
    int *ptr_nr;

    ptr_letra = letra;
    ptr_nr = nr;

    printf("Visualizando como funciona a aritmética de ponteiros\n");

    printf("\nmatriz letra = a, e, i, o, u\n");
    printf("matriz nr = 30,12,67,13,41\n");

    printf("\nVerificando o tamanho dos tipos de dados\n");
    printf("tamanho do tipo de dado char   = %d\n",sizeof(char));
    printf("tamanho do tipo de dado int    = %d\n",sizeof(int));

    printf("\nPonteiro para letra aponta para %c no endereço
%x\n",*ptr_letra,ptr_letra);
    printf("Ponteiro para nr aponta para %d no endereço %x\n",*ptr_nr,ptr_nr);

    printf("\nIncrementando os ponteiros\n");
    printf("ptr_letra + 3, ptr_nr + 2\n");
```

```
   ptr_letra += 3;
   ptr_nr += 2;

   printf("\nPonteiro para letra agora aponta para %c no endereço
%x\n",*ptr_letra,ptr_letra);
   printf("Ponteiro para nr agora aponta para %d no endereço
%x\n",*ptr_nr,ptr_nr);

   return(0);
 }
```

13.9 Exibindo uma string usando um ponteiro

Uma string é uma matriz de caracteres. Podemos usar um ponteiro para exibí-la assim:

```
/* exibindo uma string usando um ponteiro */

#include <stdio.h>

int main()
 {
   char string[40] = "Exibindo uma string usando um ponteiro.";
   char *ptr_str;

   /* Apontando para a string */
   ptr_str = string;
   printf("Apontando para o inicio da string => ptr_str = %c\n\n",*ptr_str);

   /* Exibindo toda a string usando o ponteiro */
   while(*ptr_str)
    {
      putchar(*ptr_str);
      ptr_str++;
    }
   printf("\n");

   return(0);
 }
```

13.10 Criando uma função que retorna um ponteiro

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

/* Função que converte uma strig para maiúsculas.
 * Esta função retorna um ponteiro
 */
char *converte_maiuscula(char *string)
 {
   char *inicio_da_str, *auxiliar;
   inicio_da_str = auxiliar = string;
   while(*string)
    {
      *auxiliar = toupper(*string);
      string++;
      auxiliar++;
    }
   return(inicio_da_str);
 }



int main()
 {
   char string[80] = "Usando uma função que retorna uma string.";

   printf("%s\n",string);

   converte_maiuscula(string);

   printf("%s\n",string);

   return(0);
 }
```

13.11 Matriz de ponteiros/strings

A linguagem C lhe permite criar matrizes de ponteiros. O uso mais comum para este tipo de matriz é conter strings. Abaixo segue um exemplo da declaração de uma matriz que armazena ponteiros para strings:

```
char *dias(7) =
{"Domingo","Segunda","Terça","Quarta","Quinta","Sexta","Sábado"};
```

OBSERVAÇÃO: Em matrizes de ponteiros para strings a linguagem C não inclui um item NULL para indicar o final da matriz, **você** tem que fazer isso.

13.12 Percorrendo uma matriz de strings com um laço for

```
/* percorrendo uma matriz de strings com um laço for */

#include <stdio.h>

int main()
  {
    char *dias[7] =
{"Domingo","Segunda","Terça","Quarta","Quinta","Sexta","Sábado"};
    int contador;

    for(contador = 0;contador < 7;contador++)
      printf("%do dia da semana = %s\n",contador+1,dias[contador]);

    return(0);
  }
```

13.13 Percorrendo uma matriz de strings com um ponteiro

```
/* percorrendo uma matriz de strings com um ponteiro */

#include <stdio.h>

int main()
  {
    char *dia[] =
{"Domingo","Segunda","Terça","Quarta","Quinta","Sexta","Sábado",0};
    char **ptr_dia;

    /* *dia é um ponteiro para uma string e
     * **ptr_dia é um ponteiro para um ponteiro para uma string
     */

    ptr_dia = dia; /* apontando ptr_dia para o início da matriz dia */

    while(*ptr_dia)
      {
        printf("%s\n",*ptr_dia);
        ptr_dia++;
```

```
    }

  return(0);

}

/* Quando você declara uma matriz de strings o compilador
 * não acrescenta um caractere NULL para indicar o final
 * da matriz como o faz com uma matriz de caracteres (strings).
 * Por isso você mesmo tem que inserir o caractere NULL para
 * indicar o final da matriz.
 *
 * Foi isso que foi feito ao inserir 0 no final da matriz dia
 */
```

13.14 Ponteiro para função

Você pode criar um ponteiro para uma função. O uso mais comum deste recurso é para passar uma função como parâmetro para outra função. A declaração de um ponteiro para uma função segue a sintaxe:

TIPO (*FUNÇÃO)();

Observe o uso disto no exemplo abaixo:

```
/* exemplificando o uso de ponteiro para uma função */

#include <stdio.h>

/* função que identifica o maior entre dois inteiros */
int maior(int nr1,int nr2)
 {
  return((nr1 > nr2) ? nr1 : nr2);
 }

/* função usa_maior(). Recebe dois inteiros e um
 * ponteiro para a função maior()
 */
int usa_maior(int x,int y, int (*maior)())
 {
  return(maior(x,y));
 }
```

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

   printf("Entre com o primeiro número: ");
   scanf("%d",&a);
   printf("Entre com o segundo número: ");
   scanf("%d",&b);
   printf("O maior entre os dois é %d\n",usa_maior(a,b,&maior));
   /* observe logo acima que usa_maior() recebe como argumentos
    * dois números inteiros e o endereço da função maior()
    * sendo, assim, coerente com sua declaração
    */

   return(0);
 }
```

14. Estruturas

As estruturas são utilizadas para agrupar informações relacionadas de tipos de dados diferentes. Digamos que você precisa controlar os seguintes dados relacionados ao estoque de um pequeno estabelecimento comercial:

código

nome do produto

quantidade estocada

valor de compra

valor a ser vendido

lucro

observacões sobre o produto

Este seria um caso para o uso de estruturas, pois relacionados a cada produto teremos dados do tipo int(código,quantidade), char(nome, observações) e float(valor de compra, valor de venda, lucro).

14.1 Declarando uma estrutura

A sintaxe para a declaração (ou criação) de uma estrutura é:

```
struct NOME_DA_ESTRUTURA
 {
    TIPO CAMPO1;
    TIPO CAMPO2;
    .....
```

```
    .....
    TIPO CAMPOn;
};
```

Para o caso exemplificado no item anterior poderíamos ter algo como:

```
struct produto
 {
   int codigo;
   char nome[50];
   int quantidade;
   float valor_compra;
   float valor_venda;
   float lucro;
   char obs[200];
 };
```

É importante observar que a declaração da estrutura não cria, ainda, uma variável. A declaração da estrutura apenas cria um novo tipo de dado. Após criar a estrutura você pode declarar variáveis do tipo de estrutura criado.

14.2 Declarando variáveis do tipo de uma estrutura criada

Após a declaração da estrutura você pode declarar variáveis do tipo da estrutura com a sintaxe:

```
struct NOME_DA_ESTRUTURA NOME_DA_VARIÁVEL;
```

Exemplo:

```
struct produto item;
```

Observe que esta sintaxe obecede a sintaxe normal para a declaração de variáveis:

```
TIPO NOME_DA_VARIÁVEL;
```

sendo o TIPO da variável, a nova estrutura criada (struct NOME_DA_ESTRUTURA).

Você também pode declarar a variável logo após a declaração da estrutura com uma sintaxe do tipo:

```
struct produto
 {
   int codigo;
   char nome[50];
   int quantidade;
   float valor_compra;
   float valor_venda;
   float lucro;
   char obs[200];
 }item;
```

<>14.3 Acessando os campos de uma estrutura

A sintaxe para acessar e manipular campos de estruturas é a seguinte:

NOME_DA_ESTRUTURA.CAMPO

Observe o código abaixo para um melhor esclarecimento:

```
/* acessando os campos de uma estrutura */

#include <stdio.h>

/* criando um novo tipo de dado "produto" */
struct produto
  {
    int codigo;
    char nome[50];
    int quantidade;
    float valor_compra;
    float valor_venda;
  };


int main()
  {
    struct produto item; /* declarando uma variável "item" do tipo "struct produto"
*/

    printf("Preenchendo a variável \"item\"\n");
    printf("Item.............:");
    fgets(item.nome,50,stdin);
    printf("Código...........:");
    scanf("%d",&item.codigo);
    printf("Quantidade......:");
    scanf("%d",&item.quantidade);
    printf("Valor de compra.:");
    scanf("%f",&item.valor_compra);
    printf("Valor de revenda:");
    scanf("%f",&item.valor_venda);
    printf("\n");
    printf("Exibindo os dados\n");
    printf("Código...........:%d\n",item.codigo);
```

```
    printf("Item.............:%s",item.nome);
    printf("Quantidade......:%d\n",item.quantidade);
    printf("Valor de compra.:%.2f\n",item.valor_compra);
    printf("Valor de revenda:%.2f\n",item.valor_venda);

    return(0);
  }
```

14.4 Acessando uma estrutura com ponteiros

Para acessar uma estrutura usando ponteiros você pode usar duas sintaxes:

(*NOME_DA_ESTRUTURA).CAMPO

ou

NOME_DA_ESTRUTURA->CAMPO

Exemplo:

```
/* acessando uma estrutura com ponteiros */

#include <stdio.h>

struct registro
  {
    char nome[30];
    int idade;
  };



altera_estrutura1(struct registro *ficha)
  {
    (*ficha).idade -= 10;
  }

altera_estrutura2(struct registro *ficha)
  {
    ficha->idade += 20;
  }

int main()
  {
    struct registro ficha;
```

```
   printf("Entre com seu nome:");
   fgets(ficha.nome,30,stdin);
   printf("Qual sua idade?");
   scanf("%d",&ficha.idade);

   printf("\nExibindo os dados iniciais\n");
   printf("Nome: %s",ficha.nome);
   printf("Idade: %d.\n",ficha.idade);

   altera_estrutura1(&ficha);

   printf("\nExibindo os dados após a primeira alteração\n");
   printf("Nome: %s",ficha.nome);
   printf("Idade: %d.\n",ficha.idade);

   altera_estrutura2(&ficha);

   printf("\nExibindo os dados após a segunda alteração\n");
   printf("Nome: %s",ficha.nome);
   printf("Idade: %d.\n",ficha.idade);

   return(0);
 }
```

15. Gerenciamento de memória

15.1 Preenchendo um intervalo de memória com uma constante byte

Para preencher um intervalo de memória utilize a função memset() . No gcc ela faz parte do arquivo de cabeçalho string.h e sua sintaxe é:

*memset(*STRING,CARACTER,BYTES)

a função acima preenche BYTES da área de memória apontada por *STRING com CARACTER . A função retorna um ponteiro para *STRING .

Ao preencher strings com esta função, pelo menos no ambiente Linux usando o gcc, a função não inseriu o caracter NULL para indicar o final da string, assim tive que fazer isso eu mesmo. Exemplo:

/* preechendo uma área de memória usando a função memset() */

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

```
int main()
  {
    char nome[10];

    memset(nome,'s',10);

    nome[10] = 0; /* inserindo o valor NULL para indicar o final da string */

    printf("%s\n",nome);

    return(0);
  }
```

<>15.2 Copiando um intervalo de memória

Para copiar um intervalo de memória use a função memcpy() . Ela faz parte do arquivo de cabeçalho string.h e sua sintaxe é:

*memcpy(*DESTINO,*ORIGEM,BYTES)

a função acima copia BYTES da área de meória *ORIGEM para a área de memória *DESTINO . A função retorna um ponteiro para *DESTINO . Exemplo:

```
/* copiando um intervalo de memória */

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

int main()
  {
    float nrs1[5] = {1.1,1.2,1.3,1.4,1.5};
    float nrs2[5] = {5,5,5,5,5};
    int contador;

    printf("valores de nrs1\n");
    for(contador = 0;contador < 5;contador++)
      printf("%.1f ",nrs1[contador]);
    printf("\n");

    printf("valores de nrs2 inicialmente\n");
    for(contador = 0;contador < 5;contador++)
      printf("%.1f ",nrs2[contador]);
```

```
   printf("\n");

   memcpy(nrs2,nrs1,sizeof(nrs1));
   /* observe o uso do operador "sizeof" para
    * determinar o tamanho da área de memória
    * a ser copiada
    */

   printf("valores de nrs2 após a cópia\n");
   for(contador = 0;contador < 5;contador++)
     printf("%.1f ",nrs2[contador]);
   printf("\n");

   return(0);
 }
```

15.3 Movendo um intervalo de memória

Para mover uma área de memória use a função memmove() que está no arquivo de cabeçalho string.h . Sua sintaxe é:

*memmove(*DESTINO,*ORIGEM,BYTES)

a função acima move BYTES da área de memória *ORIGEM para a área de memória *DESTINO . A função retorna um ponteiro para *DESTINO . A diferença entre esta função e a função memcpy() mostrada na seção anterior é que com esta as área de memória *DESTINO e *ORIGEM podem se sobrepor. Exemplo:

/* movendo um intervalo de memória */

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

int main()
 {
   float nrs[5] = {1.1,1.2,1.3,1.4,1.5};
   float *ptr_nrs;
   int contador,bytes;

   printf("valores de nrs inicialmente\n");
   for(contador = 0;contador < 5;contador++)
     printf("%.1f ",nrs[contador]);
   printf("\n");
```

```
    ptr_nrs = nrs;
    ptr_nrs++;

    bytes = (sizeof(float) * 4);

    memmove(nrs,ptr_nrs,bytes);

    printf("valores de nrs após a movimentação\n");
    for(contador = 0;contador < 5;contador++)
      printf("%.1f ",nrs[contador]);
    printf("\n");

    return(0);
  }
```

15.4 Copiando até encontrar um byte específico

Usando a função memccpy() , que está no arquivo de cabeçalho string.h você pode copiar um intervalo de memória até encontrar um byte específico. A sintaxe da função é;

*memccpy(*DESTINO,*ORIGEM,CARACTER,BYTES)

Ela copia, no máximo, BYTES de *ORIGEM para *DESTINO , parando se CARACTER for encontrado. Ela retorna um ponteiro para o próximo byte após CARACTER em *DESTINO ou NULL se CARACTER não for encontrado nos primeiros BYTES de *ORIGEM . Exemplo:

```
/* copiando um intervalo de memória até encontrar um
 * byte específico */

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

int main()
 {
    char frase[80] = "E como dizia meu velho professor de física...estudem!!!";
    char copia[80];
    char *ptr_copia;

    /* "ptr_copia" aponta para o resultado de memccpy().
     * Se o caracter "." for encontrado em "frase",
     * será efetuada a cópia da área de memória até
```

```
    * este caracter e "ptr_copia" apontará para o
    * próximo byte após "." em "copia.
    * Caso contrário, serão copiados 80 bytes e
    * "ptr_copia" receberá o valor NULL.
    */
   ptr_copia = memccpy(copia,frase,'.',80);

   if(ptr_copia)
     *ptr_copia = 0;

   printf("%s\n",copia);

   return(0);
 }
```

<>15.5 Comparando duas áreas de memória

Você pode comparar duas áreas de memória usando a função memcmp() . Ela faz parte do arquivo de cabeçalho string.h e sua sintaxe é:

memcpy(*AREA1,*AREA2,BYTES)

Ela compara os primeiros BYTES de *AREA1 e *AREA2 .Se *AREA1 for maior ela retorna um inteiro maior que zero, se *AREA2 for maior ela retorna um inteiro menor que zero e caso as duas áreas sejam iguais ela retorna zero. Exemplo:

/* comparando duas áreas de memória */

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

int main()
 {
   char string1[7] = "aeiou";
   char string2[7] = "Aeiou";
   int resultado;

   printf("string1 = %s\n",string1);
   printf("string2 = %s\n\n",string2);

   resultado = memcmp(string1,string2,7);

   if(resultado > 0)
```

```
    printf("string1 é maior\n");
  else if(resultado < 0)
    printf("string2 é maior\n");
  else
    printf("string1 e string2 são iguais\n");

  return(0);
}

 /* Faça algumas substituições no código
  * trocando o "A" maiúsculo de "string2"
  * para "string1" e/ou substituindo-o
  * por um "a" e observe os vários
  * resultados do programa.
  */
```

<>15.6 Trocando bytes adjacentes

Quando você precisar trocar os bytes adjacentes, normalmente para trocar dados entre máquinas que possuem ordenamento de bytes alto/baixo diferentes, use a função swab() . Ela faz parte do arquivo de cabeçalho string.h e sua sintaxe é:

swab(*ORIGEM,*DESTINO,BYTES)

Ela copia BYTES de *ORIGEM para *DESTINO trocando os bytes adjacentes, pares e ímpares. Exemplo:

```
/* trocando bytes adjacentes */

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

int main()
 {
   char str_origem[30] = "Trocando bytes adjacentes";
   char str_destino[30];

   memset(str_destino,'s',30);

   str_destino[30] = 0;

   printf("str_origem  => %s\n",str_origem);
   printf("str_destino => %s\n",str_destino);
```

```
   printf("chamando swab()\n");

   swab(str_origem,str_destino,30);

   printf("str_origem  => %s\n",str_origem);
   printf("str_destino => %s\n",str_destino);

   return(0);
 }
```

15.7 Alocando memória dinamicamente

Quando você declara uma matriz o compilador aloca memória para o armazenamento dos dados desta matriz. Caso você queira alterar o tamanho da matriz terá que modificar o código e recompilar o programa. Para evitar isso você pode alocar memória dinamicamente, ou seja, o programa aloca a memória necessária durante a execução. Para isso você pode usar as funções calloc() e malloc() . Em ambiente linux, usando o gcc, elas fazem parte do arquivo de cabeçalho stdlib.h .

A sintaxe de calloc() é:

*calloc(QUANTIDADE_DE_ELEMENTOS,TAMANHO)

Ela aloca memória para uma matriz com o número de elementos igual a QUANTIDADE_DE_ELEMENTOS , tendo cada um destes elementos TAMANHO bytes. Esta função retorna um ponteiro para o ínicio do bloco de memória alocado ou NULL caso ocorra algum problema.

Exemplo do uso de calloc() :

```
/* alocando memória dinamicamente com calloc() */

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

int main()
 {
   int *matriz1;
   int nr_elementos_matriz1;
   int contador;

   printf("Alocando memória dinamicamente usando calloc()\n");

   printf("Entre com a quantidade de elementos da matriz de inteiros :");
   scanf("%d",&nr_elementos_matriz1);
```

```
   matriz1 = (int *) calloc(nr_elementos_matriz1,sizeof(int));
   /* Observe acima a instrução "(int *)". Isto é usado
    * para converter o ponteiro retornado por calloc()
    * para um ponteiro do tipo desejado. Caso você
    * estivesse alocando memória para um ponteiro de
    * ponto flutuante esta instrução seria "(float *").
    */

   if(matriz1)
    {
      for(contador = 0;contador < nr_elementos_matriz1;contador++)
       {
         printf("Entre com o %do elemento :",contador + 1);
         scanf("%d",&matriz1[contador]);

       }

      printf("\n");

      for(contador = 0;contador < nr_elementos_matriz1;contador++)
        printf("Exibindo o %do elemento : %d\n",contador + 1,matriz1[contador]);
    }
   else
    printf("Erro ao alocar memória.\n");

   return(0);
 }
```

A sintaxe de malloc() é:

*malloc(BYTES)

Esta função aloca BYTES e retorna um ponteiro para o início da memória alocada ou NULL caso ocorra algum problema.

Exemplo do uso de malloc() :

```
/* alocando memória dinamicamente com malloc() */

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

```
int main()
 {
   int *matriz1;
   int contador;
   int nr_elementos_matriz1;

   printf("Alocando memória dinamicamente usando malloc()\n");

   printf("Entre com a quantidade de elementos da matriz de inteiros :");
   scanf("%d",&nr_elementos_matriz1);

   matriz1 = (int *) malloc(nr_elementos_matriz1 * sizeof(int));
    /* Observe acima a instrução "(int *)". Isto é usado
    *  para converter o ponteiro retornado por malloc()
    *  para um ponteiro do tipo desejado. Caso você
    *  estivesse alocando memória para um ponteiro de
    *  ponto flutuante esta instrução seria "(float *").
    */

   if(matriz1)
    {
      for(contador = 0;contador < nr_elementos_matriz1;contador++)
       {
         printf("Entre com o %do elemento :",contador + 1);
         scanf("%d",&matriz1[contador]);

       }

      printf("\n");
      for(contador = 0;contador < nr_elementos_matriz1;contador++)
        printf("Exibindo o %do elemento : %d\n",contador + 1,matriz1[contador]);
    }
   else
     printf("Erro ao alocar memória.\n");

   return(0);
 }
```

15.8 Liberando memória

Quando seu programa não precisar mais da memória alocada com as funções malloc() e calloc() ele deve liberar a memória. Para isso você pode usar a função free() , que pertence ao arquivo de cabeçalho stdlib.h e cuja sintaxe é:

free(*PONTEIRO)

Exemplo do uso de free() :

```
/* liberando memória anteriormente alocada */

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

int main()
 {
   int *matriz1;
   int contador;
   int nr_elementos_matriz1;

   printf("Alocando memória dinamicamente usando malloc()\n");
   printf("e liberando a memória alocada com free()\n");

   printf("Entre com a quantidade de elementos da matriz de inteiros :");
   scanf("%d",&nr_elementos_matriz1);

   matriz1 = (int *) malloc(nr_elementos_matriz1 * sizeof(int));

   if(matriz1)
    {
      for(contador = 0;contador < nr_elementos_matriz1;contador++)
       {
         printf("Entre com o %do elemento :",contador + 1);
         scanf("%d",&matriz1[contador]);

       }
      printf("\n");

      for(contador = 0;contador < nr_elementos_matriz1;contador++)
        printf("Exibindo o %do elemento : %d\n",contador + 1,matriz1[contador]);
```

```
      free(matriz1); /* liberando a memória alocada */
    }
  else
    printf("Erro ao alocar memória.\n");

  return(0);
 }
```

15.9 Alterando o tamanho da memória alocada

Para alterar o tamanho de um bloco de memória anteriormente alocado use a função realloc() . Ela faz parte do arquivo de cabeçalho stdlib.h e sua sintaxe é:

*realloc(*PONTEIRO,BYTES)

Ela altera o tamanho do bloco apontado por *PONTEIRO para BYTES . A memória recém-alocada não será inicializada. Se *PONTEIRO for NULL, a chamada será equivalente a malloc(BYTES) ; se BYTES for igual a zero, a chamada será equivalente a free(*PONTEIRO) . A menos que *PONTEIRO seja NULL, ele precisa ter sido retornado por uma chamada anterior para malloc() , calloc() ou realloc() .

Exemplo do uso de realloc() :

```
/* alterando um bloco de memória anteriormente alocado */

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

int main()
 {
  char *frase;

  frase = (char *) malloc(10);

  if(frase)
    printf("10 bytes alocados com sucesso\n");
  else
    printf("Erro ao alocar memória\n");

  printf("\n");

  realloc(frase,1000);

  if(frase)
```

```
    printf("1000 bytes alocados com sucesso\n");
  else
    printf("Erro ao alocar memória\n");

  return(0);
}
```

16. Argumentos da linha de comando

Seu programa C pode manipular os argumentos passados na linha de comando. Para isso existem dois parâmetros da função `main()` . Um é o parâmetro `argc` . Ele recebe o número de argumentos passados na linha de comando, incluindo o nome do programa. O segundo é `argv` . Este é uma matriz de ponteiros para strings que armazena ponteiros que apontam para cada um dos argumentos passados. Para um melhor entendimento observe o código abaixo:

```
/* entendendo os argumentos da linha de comando */

#include <stdio.h>

/* "argc" armazena o número de argumentos passados
 * na linha de comando, incluindo o nome do programa.
 * "argv" é uma matriz de ponteiros para strings e
 * armazena ponteiros que apontam para cada um
 * dos argumentos passados para a linha de comando.
 */
int main(int argc, char *argv[])
 {
  int contador;

  printf("Foram passados %d argumentos na linha de comando.\n",argc);

  for(contador = 0;contador < argc;contador++)
    printf("%do argumento => %s\n",contador,argv[contador]);

  return(0);
 }
```

Compilei o código acima com o nome de `args` . Observe abaixo como ficou uma sessão de execução deste programa:

```
[samu@pitanga c]$ ./args argumento1 argumento2
```

Foram passados 3 argumentos na linha de comando.

0o argumento => ./args

1o argumento => argumento1

2o argumento => argumento2

[samu@pitanga c]$

Caso o argumento a ser passado seja uma frase você deve colocá-lo entre aspas. Observe:

[samu@pitanga c]$ ./args "este é o primeiro argumento" argumento2 argumento3

Foram passados 4 argumentos na linha de comando.

0o argumento => ./args

1o argumento => este é o primeiro argumento

2o argumento => argumento2

3o argumento => argumento3

[samu@pitanga c]$

Você pode ainda manipular argv como ponteiro em vez de tratá-lo como uma matriz de strings. Para isso você deve declará-lo como um ponteiro para um ponteiro de strings. Exemplo:

```
/* tratando "argv" como ponteiro */

#include <stdio.h>

/* observe a declaração de "argv" como
 * um ponteiro para um ponteiro
 */
int main(int argc, char **argv[])
 {
   int contador = 0;

   printf("Foram passados %d argumentos na linha de comando.\n",argc);

   while(*argv)
    {
      printf("%do argumento => %s\n",contador,*argv);
      contador++;
      argv++;
    }

   return(0);
```

```
}
```

Não que esqueça que argv é uma matriz de strings, então os dados passados na linha de comando são todos considerados caracteres ASCII. Isto significa que os valores numéricos deverão ser convertidos. Para maiores detalhes sobre a conversão de strings para valores numéricos dê uma olhada na seção Convertendo strings em números.

17. Manipulando o ambiente

17.1 Exibindo as variáveis de ambiente

Além de argc e argv a função main possui o parâmetro env que lhe permite manipular as variáveis de ambiente. env é uma matriz de ponteiros para strings que armazena ponteiros para cada uma das variáveis de ambiente. Abaixo segue um código que exibe as variáveis de ambiente do sistema:

```
/* exibindo as variáveis de ambiente */


#include <stdio.h>


int main(int argc,char *argv[],char *env[])
 {
   int contador;
   printf("Exibindo as variáveis de ambiente\n");
   printf("==================================\n");
   for(contador = 0;env[contador] != NULL;contador++)
     printf("env[%d] => %s\n",contador,env[contador]);


   return(0);
 }
```

Você pode tratar env como um ponteiro em vez de tratá-lo como uma matriz de strings. Para isso você deve declará-lo como um ponteiro para um ponteiro de strings. Exemplo;

```
/* tratando "env" como ponteiro */


#include <stdio.h>


int main(int argc,char *argv[],char **env[])
 {
   int contador = 0;


   printf("Exibindo as variáveis de ambiente\n");
   printf("==================================\n");
```

```
   while(*env)
     printf("env[%d] => %s\n",contador++,*env++);

   return(0);
 }
```

17.2 Pesquisando uma variável no ambiente

Para pesquisar um item no ambiente use a função getenv() . Ela faz parte do arquivo de cabeçalho stdlib.h e sua sintaxe é:

getenv("NOME")

Ela procura pela variável NOME na lista de variáveis de ambiente. Caso encontre ela retorna um ponteiro para a variável, caso contrário ela retorna NULL. Exemplo:

```
/* pesquisando uma variável de ambiente */

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

int main()
 {
   char *variavel;

   variavel = getenv("USER");

   if(*variavel)
     printf("USER = %s\n",variavel);
   else
     printf("A variável de ambiente USER não está definida.\n");

   return(0);
 }
```

17.3 Alterando o valor ou adicionando uma variável ao ambiente

Para alterar o valor ou adicionar uma variável ao ambiente use a função putenv() . Ela faz parte do arquivo de cabeçalho stdlib.h e sua sintaxe é:

putenv("VARIAVEL=novo_valor")

Caso VARIAVEL exista no ambiente seu valor é alterado para novo_valor , caso não exista esta é adicionada ao ambiente. A função retorna zero caso tudo ocorra bem, caso contrário ela retorna -1. Exemplo:

```
/* adicionando uma variável ao ambiente */
```

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

int main(int argc,char *argv[],char *env[])
 {
   int contador;
   char *variavel;

   if(putenv("GURU=samu"))
     printf("Erro ao adicionar a variável de ambiente GURU\n");
   else
    {
      variavel = getenv("GURU");
      if(*variavel)
        printf("GURU = %s\n",variavel);
      else
        printf("A variável de ambiente GURU não está definida.\n");
    }

   return(0);
 }
```

18. Encerrando seu programa

18.1 return

O comando return(VALOR) encerra uma função retornando VALOR para a função chamadora. Caso a função chamadora seja main() , o programa será encerrado e o valor retornado passado para o sistema operacional. Exemplo:

```
/* verificando o funcionamento de "return" */

#include <stdio.h>



int soma(int a, int b)
 {
   int resultado;
   resultado = a + b;
   /* "resultado" será passado a função chamadora
    * no caso "printf" */
   return(resultado);
```

```
 }


int main()
 {
   int nr1,nr2,valor;

   printf("Entre com o valor do primeiro número :");
   scanf("%d",&nr1);
   printf("Entre com o valor do segundo número :");
   scanf("%d",&nr2);

   /* Chamando a função soma. O comando "return"
    * da função soma retornará "resultado" para
    * aqui. */
   printf("A soma dos dois é %d\n",soma(nr1,nr2));

   printf("Entre com o valor a ser retornado para o sistema operacional :");
   scanf("%d",&valor);
   printf("Em ambiente Linux, digite o comando shell\n\n");
   printf("$  echo $?\n\n");
   printf("Para ver o valor retornado pelo programa\n");

   /* "valor" será retornado para o sistema operacional
    * pois este foi a função chamadora de "main" */
   return(valor);
 }
```

18.2 exit

Ao ser chamado, o comando exit(VALOR) encerra imediatamente o programa e retorna VALOR para o sistema operacional. Exemplo:

```
/* verificando o funcionamento de "exit" */

#include <stdio.h>


int soma(int a, int b)
 {
   int resultado;
```

```
    resultado = a + b;
    /* "exit" encerra o programa imediatamente e retorna
     * "resultado" para o sistema operacional */
    exit(resultado);
  }


int main()
  {
    int nr1,nr2,valor;

    printf("Entre com o valor do primeiro número :");
    scanf("%d",&nr1);
    printf("Entre com o valor do segundo número :");
    scanf("%d",&nr2);

    /* Chamando a função soma. O comando "return"
     * da função soma retornará "resultado" para
     * aqui. */
    printf("A soma dos dois é %d\n",soma(nr1,nr2));

    /* Nada abaixo disto será executado pois o comando "exit"
     * chamado na função "soma" acima encerrará imediatamente
     * o programa e retornará "resultado" para o sistema operacional
     */
    printf("Entre com o valor a ser retornado para o sistema operacional :");
    scanf("%d",&valor);
    printf("Em ambiente Linux, digite o comando shell\n\n");
    printf("$  echo $?\n\n");
    printf("Para ver o valor retornado pelo programa\n");

    /* "valor" será retornado para o sistema operacional
     * pois este foi a função chamadora de "main" */
    return(valor);
  }
```

18.3 abort

Ao encontrar a função abort o programa é imediatamente encerrado e um valor de erro padrão do sistema é retornado para o sistema operacional. Exemplo:

```
/* verificando o funcionamento de "abort" */

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

int soma(int a, int b)
  {
    int resultado;
    resultado = a + b;
    /* Ao encontrar "abort" o programa é imediatamente
     * encerrado sendo retornado para o sistema
     * operacional um dos valores de erro padrão
     * do sistema */
    abort();
    exit(resultado);
  }


int main()
  {
    int nr1,nr2,valor;

    printf("Entre com o valor do primeiro número :");
    scanf("%d",&nr1);
    printf("Entre com o valor do segundo número :");
    scanf("%d",&nr2);

    /* Chamando a função soma. O comando "return"
     * da função soma retornará "resultado" para
     * aqui. */
    printf("A soma dos dois é %d\n",soma(nr1,nr2));

    /* Nada abaixo disto será executado pois o comando "exit"
     * chamado na função "soma" acima encerrará imediatamente
     * o programa e retornará "resultado" para o sistema operacional
     */
    printf("Entre com o valor a ser retornado para o sistema operacional :");
    scanf("%d",&valor);
```

```
   printf("Em ambiente Linux, digite o comando shell\n\n");
   printf("$  echo $?\n\n");
   printf("Para ver o valor retornado pelo programa\n");

   /* "valor" será retornado para o sistema operacional
    * pois este foi a função chamadora de "main" */
   return(valor);
 }
```

18.4 Definindo funções a serem executadas no encerramento do programa

Para definir funções a serem executadas no encerramento do programa use a função atexit() . Ela faz parte do arquivo de cabeçalho stdlib.h e sua sintaxe é:

atexit(FUNÇÃO)

ela faz com que FUNÇÃO seja chamada quando o programa terminar normalmente, ou seja, com uma chamada a exit() ou com uma chamada a return() a partir de main() . FUNÇÃO não recebe argumentos, assim, caso ela precise acessar dados estes devem ter sido declarados como variávesi globais. Você pode ainda definir mais de uma função a ser chamada com atexit . Neste caso elas serão chamadas na ordem inversa da declaração, ou seja, a última função declarada será a primeira a ser chamada. Caso tudo ocorra bem atexit retorna zero, caso contrário retorna -1. Observe abaixo um exemplo do uso de atexit :

```
/* utilizando "atexit" */

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

void primeira()
 {
   printf("Primeira função chamada por atexit\n");
 }

void segunda()
 {
   printf("Segunda função chamada por atexit\n");
 }

void terceira()
 {
   printf("Terceira função chamada por atexit\n");
 }
```

```
int main()
  {
    printf("Chamando atexit\n\n");
    atexit(primeira);
    atexit(segunda);
    atexit(terceira);

    return(0);
  }
```