Universidade de Passo Fundo
Instituto de Ciências Exatas e Geociências
Curso de Ciência da Computação

Linguagem C

CCC011 - Programação I
Prof. Marcos José Brusso <brusso@upf.br>
http://vitoria.upf.br/~brusso/progc
Semestre: 2003/2

Sumário

1. Introdução

1.1. Estrutura do programa C

Um programa em linguagem C é formado por uma ou mais funções. Cada função é um bloco de código delimitado que deve ter um nome e executa um conjunto de operações. Uma função denominada **`main`** é obrigatória em todos os programas, pois é o seu ponto de entrada, isto é, o programa começa a ser executado no início da função **`main`** e termina ao final desta função. Normalmente a função **`main`** inicia com a declaração **`int main(void)`** e tem seu corpo delimitado por um par de chaves `{}`.

Ao concluir a função **`main`,** com o comando **`return`**, a execução do programa é finalizada, sendo que pelo padrão ANSI, esta função deve retornar 0 (zero) se o programa foi finalizado com sucesso, ou um valor maior que zero caso ele tenha sido finalizado por uma situação de erro. Além da função main, o programa pode possuir outras funções, como será visto adiante (pg. 19), sendo que estas devem ser, direta ou indiretamente, chamadas pela função **`main`**.

Na codificação de um programa, deve-se observar que o compilador C diferencia letras maiúsculas de minúsculas, portanto **TESTE** é diferente de **teste**, que por sua vez é diferente de **Teste**. Todas as palavras reservadas da linguagem devem ser digitadas em letra minúscula.

```
/* prog01.c
   Exemplo de programa em C
*/

#include <stdio.h>

int main(void)
{
   printf("Programa C muito simples\n");
   getchar();              /* Aguarda pressionar Enter */
   return(0);
}
```

1.2. Comentários

Os comentários servem principalmente para documentação do programa e são ignorados pelo compilador, portanto não irão afetar o programa executável gerado. Os comentário iniciam com o símbolo **`/*`** e se estendem até aparecer o símbolo **`*/`**. Um comentário pode aparecer em qualquer lugar no programa onde possa aparecer um espaço em branco e pode se estender por mais de uma linha.

1.3. Diretiva #include

Toda a diretiva, em C, começa com o símbolo **`#`** no início da linha. Esta diretiva inclui o conteúdo de um outro arquivo dentro do programa atual, ou seja, a linha que contêm a diretiva é substituída pelo conteúdo do arquivo especificado.

Sintaxe:

```
#include <nome do arquivo>
```
ou
```
#include "nome do arquivo"
```

O primeiro caso é o mais utilizado. Ele serve para incluir alguns arquivos que contêm declaração das funções da biblioteca padrão, entre outras coisas. Estes arquivos, normalmente, possuem a extensão .h e se encontram em algum diretório pré-definido pelo compilador (`/usr/include` no Linux; `c:\dev-c++\include` no Windows com o Dev-C++). Sempre que o programa utilizar alguma função da biblioteca-padrão deve ser incluido o arquivo correspondente. A tabela a seguir apresenta alguns dos principais .h do C:

Arquivo	Descrição
`stdio.h`	Funções de entrada e saída (I/O)
`string.h`	Funções de tratamento de strings
`math.h`	Funções matemáticas
`ctype.h`	Funções de teste e tratamento de caracteres
`stdlib.h`	Funções de uso genérico

A segunda forma, onde o nome do arquivo aparece entre aspas duplas, serve normalmente para incluir algum arquivo que tenha sido criado pelo próprio programador ou por terceiros e que se encontre no diretório atual, ou seja, no mesmo diretório do programa que está sendo compilado.

2. Armazenamento de dados

2.1. Variáveis

Os dados que são processados pelos programas são armazenados na memória em **variáveis**. Em C a declaração das variáveis antes do seu uso é obrigatória, quando é definido o seu tipo, o seu nome, e eventualmente, o seu valor inicial, como será visto na seqüência. O tipo da variável vai definir o tamanho em bytes que ela ocupará na memória do computador e o intervalo de valores que ela poderá armazenar. O primeiro caracter no nome de uma variável deverá ser uma letra ou *underline* (_) e os demais caracteres podem ser letras, números ou *underlines*. A declaração das variáveis deverá ser feita antes de qualquer comando executável.

Um programa pode ter **variáveis globais**, que são definidas fora de qualquer função e que estarão disponíveis para serem utilizadas em todo o programa, ou **variáveis locais** que são declaradas dentro de uma função e são visíveis apenas na função em que foram criadas. Funções distintas podem ter variáveis locais com o mesmo nome. Se uma função declarar uma variável com o mesmo nome de uma variável global, esta variável local ocultará aquela global, que não poderá ser acessada.

Exemplo:

```
/* prog02.c */
int k, x;             /* 2 variáveis globais do tipo int */

int main(void)
{
      int x, y, z;    /* 3 variáveis locais do tipo int   */
      double v;       /* 1 variável local do tipo double */

      x = 5;          /* Está atribuindo para a local */

      return(0);
}
```

2.2. Tipos de dados

A linguagem C disponibiliza quatro tipos básicos que podem ser utilizados para a declaração de variáveis:

Tipo	Descrição	Bytes	Intervalo de valores
`char`	Um único caracter	1	0 a 255
`int`	Números inteiros (sem casas decimais)	4	-2147483648 a +2147483647
`float`	Números em ponto flutuante com precisão simples (7 casas decimais)	4	$3{,}4.10^{-38}$ a $3{,}4.10^{38}$
`double`	Números em ponto flutuante com precisão dupla (15 casas decimais)	8	$3{,}4.10^{-4932}$ a $1{,}1.10^{-4932}$

2.3. Inicialização

Inicialização corresponde à definir um valor inicial para a variável, o qual será armazenado no momento em que a variável é criada. A inicialização de uma variável pode ser feita com uma constante, expressão ou função.

Exemplo:

```
int a=10, b=a-50, c=abs(b);
char letra='A';
float vf=25.781;
```

Na falta de inicialização, variáveis globais são inicializadas automaticamente com zero. As variáveis locais possuem valor indefinido (lixo) quando não inicializadas na declaração, portanto, não podem ter seus valores utilizados em nenhuma operação antes que algum conteúdo seja atribuído, sob pena de que estas operações vão resultar em valores incorretos.

2.4. Diretiva #define

Esta diretiva é utilizada para a definição de *macros,* que são nomes que representam valores constantes. Após a definição da macro, toda ocorrência do seu nome no programa é substituída pelo valor definido. Por convenção todo o nome de macro é escrito em letras maiúsculas.

Sintaxe:
```
#define nome valor
```

Exemplo:
```
#define PI 3.14159265358979323846
...
area = PI * (raio*raio);
```

2.5. Arranjos: vetores e matrizes

Arranjos (ou *arrays*) são conjuntos de valores de um mesmo tipo, com um nome em comum e que são acessados através de um índice. O índice indica a posição do valor a ser manipulado dentro do arranjo, sendo que, em C, todos os arranjos iniciam com o primeiro elemento na posição 0 (zero). O número indicado entre colchetes, na declaração, corresponde ao tamanho do arranjo, isto é, o seu número de elementos. Desta forma, a declaração `int vet[10]` corresponde a um arranjo com 10 valores,estando o primeiro elemento na posição `[0]` e último na posição `[9]`. A linguagem C não faz a verificação automática dos limites em um arranjo, sendo isto responsabilidade do programador.
Os arranjos podem ter uma ou mais dimensões. Os unidimensionais são normalmente conhecidos por *vetores*, enquanto que os arranjos com mais de uma dimensão são chamados de *matrizes*. Cada um dos índices correspondente à cada dimensão de uma matriz deve estar delimitado por um par de colchetes. Numa matriz com duas dimensões, a primeira corresponde à *linha* e a segunda corresponde à *coluna*.

Exemplo:
```
#define TAMANHO  15

int ivetor[100];     /* Vetor com 100 elementos: [0]..[99] */
float fmatriz[2][5]; /* Matriz 2 linhas:[0]..[1] e 5 colunas:[0]..[4] */
double dvetor[TAMANHO]; /* Vetor com 15 elementos: [0]..[14] */
```

2.5.1. Inicialização de arranjos

Os arranjos podem ser inicializados na declaração atribuindo-se um conjunto de valores delimitados por um par de chaves e separados por vírgula. Se o vetor estiver sendo inicializado, o seu tamanho pode ser omitido, sendo então calculado pelo compilador. Para a inicialização de matrizes, cada uma das dimensões deve estar delimitada por um par de chaves, inserindo-se ainda um par de chaves para toda a matriz.

Exemplo:
```
int va[5] = {2, 4, 6, 8, 10};
int vb[]={76, 0, 18};
int matriz[2][4] = {{9, 4, 100, -9},{4, 13, 50, 27}};
```

2.5.2. Strings

Strings são seqüências de caracteres utilizadas para o armazenamento de texto. Não existe em C um tipo específico para o armazenamento de strings, elas são simplesmente vetores de caracteres (`char`). Uma particularidade das strings, em C, é que elas possuem um caracter terminador, o qual delimita o final do seu conteúdo, que é o caracter `'\0'` (lê-se "contrabarra zero"). Desta forma, o tamanho da string deve ser definido com um caracter a mais do que será efetivamente necessário para o programa.
O conteúdo de uma string pode ser inicializado por uma seqüência de caracteres entre aspas duplas . Neste caso o compilador C coloca automaticamente o `'\0'` no final.

Exemplo:
```
char estado[3]="RS";            /* 3 caracteres: 'R', 'S' e '\0' */
char cidade[100]="Passo Fundo"; /* string com 100 caracteres     */
char nome[]="Fulano de Tal";    /* string com 14 caracteres      */
```

3. Entrada e saída padrão

As principais funções para entrada e saída (leitura e escrita) de valores, em C, estão definida em `<stdio.h>` O nome deste arquivo originou-se do termo inglês "**st**an**d**ard **i**nput/**o**utput", que significa "entrada e saída padrão".

3.1. A função printf

A função `printf` é uma função da biblioteca padrão utilizada para escrever valores. Ela pode receber diversos parâmetros, sendo que o primeiro deverá ser uma string, chamada **string de formato**. Somente será exibido o que for solicitado dentro desta string. Tudo o que aparecer nesta string que não é um especificador de formato será impresso literalmente. Cada especificador de formato encontrado nesta string é substituído pelo parâmetro seguinte na lista de parâmetros da função.

3.1.1. Especificadores de formato

São simbolos, representados por um `%` mais um outro caracter, que indicam que um valor vai ser escrito pela função, na posição correspondente da string de formato. Indicam também o tipo e/ou a base numérica do dado a ser escrito. Cada valor deve ter um especificador correspondente: se faltarem especificadores, os últimos valores não serão exibidos; se sobrarem especificadores, serão exibidos valores indeterminados (lixo). A tabela a seguir apresenta os principais especificadores de formato utilizados com a função `printf`.

Formato	Descrição
`%d`	Número inteiro decimal
`%o`	Número inteiro octal
`%x`	Número inteiro hexadecimal
`%c`	Um único caracter
`%s`	String
`%f`	Número em ponto flutuante
`%%`	Exibe um %

3.1.2. O tamanho e precisão do campo

O tamanho do campo indica quantas posições da saída serão utilizados para exibir o valor correspondente à um especificador de formato. O tamanho do campo é determinado colocando-se um número entre o `%` e o caracter indicador do formato. Por exemplo, **`%5d`** indica que o valor a ser impresso vai ocupa 5 posições na tela. Por padrão, as posições em excesso são preenchidas com brancos e o valor é alinhado à direita do campo.

Se o tamanho do campo for precedido por um símbolo **`-`** (menos), então o valor será alinhado à esquerda do campo. Se o tamanho do campo for precedido com o símbolo `0` (zero), então as posições excedentes são preenchidas com zeros. Sempre que o tamanho do campo for insuficiente para exibir o valor solicitado, este tamanho é ignorado. Desta forma um valor nunca é truncado.

Para valores em ponto-flutuante, pode-se definir o número de casas decimais a serem impressas, colocando-se o um ponto (.) e outro número depois do tamanho do campo. Por exemplo, o formato **`%10.2f`**, exibe o valor em ponto-flutuante com duas casas decimais, dentro de um campo de tamanho 10. Se esta mesma lógica for utilizada com strings (formato **`%s`**), o valor antes do ponto continua indicando a tamanho do campo, porém a segunda parte limitará o número máximo de caracteres da string a serem impressos.

Exemplo:

```
/* prog03.c */
#include <stdio.h>

int main(void)
{
   char letra = 'A';
   int num = 15;
   double dv = 13.71;
   char curso[]="COMPUTACAO";

   printf("[%c]", curso[1]);
   printf("[%c] [%d]\n",letra, letra);
   printf("[%d] [%o] [%x]\n", num, num, num);
   printf("[%5d] [%05d] [%-5d]\n",num, num, num);
   printf("[%7.1f]\n",dv);
   printf("[%.3f]\n",dv);
   printf("[%-7.1f]\n",dv);
   printf("[%15s]\n", curso);
   printf("[%-15s]\n", curso);
   printf("[%15.4s]\n", curso);
   printf("[%-15.4s]\n", curso);
   getchar();
   return(0);
}
```

3.2. Função getchar

Esta função lê um caracter e retorna o caracter digitado, que pode ser atribuído para uma variável `char`. Deve-se teclar **Enter** após o caracter a ser lido.

Exemplo:

```
char letra;
letra = getchar();
```

Ela também pode ser utilizada simplesmente para dar uma pausa no programa até que o usuário tecle **Enter**. Neste caso simplesmente descarta-se o valor de retorno, não atribuido-o à nenhuma variável.

Exemplo:

```
getchar();
```

3.3. Função scanf

É a principal função de entrada de dados da biblioteca padrão, sendo utilizada principalmente para a leitura de valoes numéricos (variáveis `int`, `float` e `double`). É utilizada de forma semelhante à função `printf`, iniciando com uma string de formato, onde deve aparecer os especificadores de formato adequados às variáveis que estão sendo lidas, conforme a tabela abaixo.

Formato	Tipo
`%d`	int
`%f`	float
`%lf`	double
`%c`	Um caracter
`%s`	Uma string. *Atenção: espaço em branco finaliza entrada.*

As variáveis que serão lidas devem ser precedidas do símbolo **`&`**, exceto para strings. Qualquer caracter inserido na string de formato que não é um especificador é considerado separador, isto é, delimita o conteúdo que será atribuído a cada uma das variáveis lida.

> **!** Devido a forma como a função `scanf` trata o *buffer* (área de armazenamento temporária) de entrada, quando ela for utilizada em conjunto com as demais funções de leitura deve-se limpar este *buffer* com `fflush(stdin);`

Exemplo:

```
/* prog04.c */
#include <stdio.h>

int main(void)
{
    int dia, mes, ano;
    float temp, far;

    printf("Informe a temperatura (Celcius): ");
    scanf("%f", &temp);
    printf("Informe a data (dd/mm/aaaa): ");
    scanf("%d/%d/%d", &dia, &mes, &ano);
    far=(9.0 / 5.0 * temp + 32.0);
    printf("Em %02d/%02d/%04d a temperatura foi %.1f (fahrenheit)\n",
                dia, mes, ano, far);
    fflush(stdin);
    getchar();
    return(0);
}
```

3.4. Função gets

Esta função pode ser utilizada para a entrada de texto em strings. Deve-se chamar a função passando-se a string a ser lida como parâmetro, isto é, entre os parênteses. Ela é mais adequada para esta situação do que a função **`scanf`**, com o formato **`%s`**, que não aceita espaços em branco no conteúdo digitado, o que não acontece com **`gets`**.

Exemplo:

```
/* prog05.c */
#include <stdio.h>

int main(void)
{
    char nome[100];

    printf("Informe o seu nome: ");
    gets(nome);
    printf("O seu nome é \"%s\"\n", nome);
    printf("A primeira letra é \'%c\'\n", nome[0]);
    getchar();
    return(0);
}
```

4. Operadores

4.1. Operadores aritméticos

São utilizados para efetuar as operações aritméticas com os seus operandos. Estes operandos podem ser utilizados com qualquer tipo de dados, exceto o resto da divisão, o qual não pode ter operandos em ponto flutuante. Atenção especial deve serdada à operação de divisão. Numa operação onde tanto o dividendo como o divisor forem valores inteiros o resultado perderá as casas decimais, independente do tipo da variável ao qual estará sendo atribuído.

Operador	Descrição
=	Atribuição
+	Soma
–	Subtração
*	Multiplicação
/	Divisão (se os dois valores forem `int`, o resultado não terá casas decimais)
%	Resto da divisão inteira

Exemplo:

```
/* prog06.c */
#include <stdio.h>

int main(void)
{
   int dividendo=10, divisor=3;
   float quociente=0.0;

   quociente = dividendo / divisor;
   printf("%d/%d = %.2f\n", dividendo, divisor, quociente);
   getchar();
   return(0);
}
```

Operador	Descrição
++	Incremento pré ou pós-fixado
--	Decremento pré ou pós-fixado

Exemplo:

```
/* prog07.c */
#include <stdio.h>

int main(void)
{
   int a,b,c;

   a=b=c=2;
   b= ++a;
   c= b++;
   printf("a:%d\nb:%d\nc:%d\n", a, b, --c);
   getchar();
   return(0);
}
```

4.2. Operadores combinados

Sempre que em um programa C aparece uma expressão onde o resultado da operação está sendo atribuída para o seu primeiro operando (da esquerda), conforme o formato `x = x op y;` esta expressão pode ser reduzida para o formato `x op= y;`

Expressão Normal	Expressão Simplificada
`a = a + b;`	`a+=b;`
`a = a - b;`	`a-=b;`
`a = a * b;`	`a*=b;`
`a = a / b;`	`a/=b;`
`a = a % b;`	`a%=b;`

4.3. Operadores relacionais

Os operadores relacionais são utilizados em expressões condicionais para a comparação do valor de duas expressões.

Operador	Descrição
>	Maior que
>=	Maior ou igual à
<	Menor que
<=	Menor ou igual à
==	Igual à
!=	Diferente de

4.4. Operadores lógicos

Os operadores lógicos são utilizados para conectar expressões lógicas sendo geralmente utilizados em expressões condicionais.

Operador	Descrição
&&	AND lógico
||	OR lógico
!	NOT lógico

4.5. Operador ternário

O nome deste operador deve-se ao fato que ele possui 3 (três) operandos. O primeiro é uma expressão condicional que será avaliada (testada). Se esta condição for verdadeira, o segundo operando é utilizado (o valor que está após o ponto de interrogação). Se a condição for falsa, será utilizado o terceiro operando (o último valor, após o dois-pontos)

Sintaxe:

```
(condição)?valor1:valor2
```

Exemplo:

```
/* prog08.c */
#include <stdio.h>

int main(void)
{
   int n1, n2, maior;

   printf("Digite dois valores:\n");
   scanf("%d\n%d", &n1, &n2);
   maior = (n1>n2)?n1:n2;
   printf("O maior e' %d\n", maior);
   fflush(stdin); getchar();
   return(0);
}
```

4.6. Operador sizeof

Este operador retorna o tamanho em bytes ocupado na memória pela expressão ou pelo tipo indicado. O tamanho de uma variável nunca depende do seu conteúdo armazenado, mas apenas do tipo com o qual ela foi declarada. O tamanho de um arranjo é igual a soma do tamanho de seus elementos.

Sintaxe:

```
sizeof(expressão) ou sizeof(tipo)
```

Exemplo:

```
int r, x=100, vet[3];
r = sizeof(x);        /* r recebe 4 (int -> 4 bytes) */
r = sizeof(double);   /* r recebe 8 (double -> 8 bytes) */
r = sizeof(vet);     /* r recebe 12 (3*4 -> 12) */
```

4.7. Operador de moldagem ou cast

Colocando-se o nome de um tipo de dados entre parênteses à esquerda de uma expressão, força-se aquela expressão a assumir o tipo indicado, isto é, converte-se o valor naquele ponto do programa. Quando utilizado com variável, o tipo dela não é modificado, apenas o seu valor é temporariamente convertido.

Sintaxe:

```
(tipo) expressão
```

Exemplo:

```
/* prog09.c */
#include <stdio.h>

int main(void)
{
   int dividendo=10, divisor=3;
   float quociente=0.0;

   quociente = (float)dividendo / divisor;
   printf("%d/%d = %.2f\n", dividendo, divisor, quociente);
   getchar();
   return(0);
}
```

5. Comandos de controle do programa

5.1. Comando if

Este comando pode ser utilizado quando é necessária a execução condicional de um determinado trecho do programa, isto é, alguns comandos somente devem ser executados se uma condição é verdadeira ou falsa.

Sintaxe:

```
if(condição){
      comandos a serem executados se a condição é verdadeira
}
else{
      comandos a serem executados se a condição é falsa
}
```

Na linguagem C a condição deve obrigatoriamente estar entre parênteses e , ao contrario de muitas linguagens de programação, não existe a palavra **`then`**. Se houverem mais do que um comando a ser executado obrigatoriamente eles devem estar delimitados por um par de chaves, enquanto que se houver um único comando, as chaves são opcionais. A parte do comando formada pelo **`else`**, com os comandos que serão executados se a condição for falsa não é obrigatória. Caso não seja utilizada uma comparação como condição, mas apenas algum valor numérico, este valor é avaliado: se o seu valor é 0 (zero) é considerado falso; qualquer outro valor é considerado verdadeiro.

```
/* prog10.c */
#include <stdio.h>

int main(void)
{
    int ano, dias=365;

    printf("Informe o ano: ");
    scanf("%d", &ano);
    if(ano%4==0){
        printf("%d eh um ano bissexto", ano);
        dias=366;
    }
    else
        printf("%d nao eh um ano bissexto", ano);

    printf(" e possui %d dias\n", dias);
    fflush(stdin); getchar();
    return(0);
}
```

5.2. Comando do while

Este comando pode ser utilizado quando um trexo de código precisa ser executado diversas vezes, repetidamente, enquanto uma determinada condição for verdadeira. Como a condição é testada após o conjunto de comandos terem sido executados, o corpo do laço é executado, pelo menos, uma vez.

Sintaxe:

```
do{
      comandos a serem repetidos
}while(condição);
```

O conjunto de comandos é executado, e ao final de cada passagem (iteração) a condição é avaliada. Se ela for verdadeira, a seqüência de execução do programa retorna ao início do laço, executando novamente os comandos. Se a condição for falsa, o laço é finalizado, prosseguindo a execução do programa no primeiro comando após o `while`.

Exemplo:

```
/* prog11.c */
#include <stdio.h>

int main(void)
{
    double n;

    printf("Informe um valor maior que zero: ");
    do{
        scanf("%lf", &n);
        if(n<=0)
            printf("Invalido! Redigite...: ");
    }while(n<=0.0);

    printf("O inverso de %f eh %f\n", n, 1/n);
    fflush(stdin); getchar();
    return(0);
}
```

5.3. Comando while

Similar ao comando anterior, o comando `while` executa um bloco de código enquanto uma condição for verdadeira. A diferença é que, neste caso, a condição está no início do laço. Como a condição é avaliada no início de cada passagem, o conjunto de comando não será executado nenhuma vez se, ao entrar neste laço, a condição for falsa.

Sintaxe:

```
while(condição){
      comandos a serem repetidos;
}
```

Ao início de cada passagem a condição é avaliada. Se ela for verdadeira, o conjunto de comandos é executado, retornando ao teste da condição assim que todo o corpo do laço tenha sido executado. Se a condição for falsa, o laço é finalizado, prosseguindo a execução do programa no comando seguinte.

Exemplo:

```
/* prog12.c
   Calcula potencias de base 2
*/

#include <stdio.h>
int main(void)
{
    int n, i, pot=1;

    printf("Informe um número positivo:");
    scanf("%d", &n);
    i=n;
    while(i > 0){
        pot*=2;
        i--;
    }
    printf("2 elevado ao expoente %d = %d\n",n, pot);
    fflush(stdin); getchar();
    return(0);
}
```

5.4. Comando continue

Este comando pode ser utilizado dentro de laços para ignorar o restante da passagem atual. Os demais comandos que ainda não foram executados dentro do laço, abaixo do **`continue`**, não serão executados e a seqüência de execução do programa passa para o teste da condição do laço atual.

Sintaxe:
```
continue;
```

5.5. Comando break

Este comando finaliza o laço atual, pulando para o primeiro comando após o final dele. O laço é abortado independente da condição ser verdadeira ou falsa, pois o **`break`** interrompe-o incondicionalmente.

Sintaxe:
```
break;
```

Exemplo:
```
/* prog13.c
   Lê e soma valores válidos até digitar 0
*/
#include <stdio.h>

int main(void)
{
      int num, i=0, soma=0;

      printf("Digite valores entre 1 e 100 (0 para parar)\n");
      while(1){               /* Sempre verdadeiro */
            printf("%d> ", i);
            scanf("%d", &num);
            if(num==0)
                  break;
            if(num<0 || num > 100){
                  printf("Redigite...\n");
                  continue;
            }
         soma+=num;
         i++;
      }
     printf("Soma dos valores informados=%d\n", soma);
     fflush(stdin); getchar();
     return(0);
}
```

5.6. Comando for

Este comando é utilizado normalmente para criar um laço que contêm um conjunto de comandos que será executado um número fixo de vezes.

Sintaxe:
```
for(inicialização; condição; atualização){
      comandos a serem repetidos;
}
```

Ele é composto por 3 partes separadas por ponto-e-vírgula. A primeira parte (*inicialização*) é onde a(s) variável(is) de controle do laço tem seu valor inicial definido. Pode-se inicializar mais de uma variável neste ponto, separando-as por vírgula. Se esta parte do comando for omitida (ficar em branco), as variáveis manterão os valores atribuídos anteriormente no programa.

A segunda parte (*condição*) contêm um teste que será executado ao início de cada passagem. Se esta condição é verdadeira, executa-se mais uma vez o corpo do **`for`**. Se a condição for falsa, o laço é finalizado. É portanto uma condição do tipo "*enquanto*", assim como todos os laços em linguagem C. Se a condição for omitida, o laço é executado indefinidamente (laço infinito) a não ser que seja interrompido por um **`break`**.

Na terceira parte do camando (após o segundo ponto-e-vírgula) deve ser fornecido um comando (para cada variável de controle do laço) que atualize o valor da variável. Normalmente é utilizado incremento ou decremento, mas pode-se utilizar qualquer expressão que modifique o valor da variável, dependendo da aplicação. Se mais que uma variável forem modificadas neste ponto, deve-se separá-las por vírgulas.

Exemplo:

```
/*  prog14.c
   Exibe o conteudo de um vetor, a soma e a media seus elementos
*/
#include <stdio.h>
#define TAMVET 5

int main(void)
{
   int i;
   float vetor[TAMVET]={3.5, 12.8, 1.75, 0.25, 100.0}, soma, media;

   for(soma=0.0,i=0; i<TAMVET; i++){
      printf("vetor[%d]: %.2f\n", i , vetor[i]);
      soma += vetor[i];
   }
   media = soma/TAMVET;
   printf("\nSoma: %.2f\nMedia: %.2f\n", soma, media);
   getchar();
   return(0);
}
```

5.7. Comando switch

Este é um comando de seleção útil para ser utilizado quando a mesma variável é comparada diversas vezes com valores diferentes. Um **`switch`** pode substituir muitos comandos **`if`**.

Sintaxe:

```
switch(variável){
      case valor1:
            comandos a serem executados se variável==valor1
            break;
      case valor2:
            comandos a serem executados se variável==valor12
            break;
      ...
      case valorn:
            comandos a serem executados se variável==valorn
            break;
      default:
            comandos a serem executados se o valor não foi encontrado
}
```

O valor da variável é comparado com o valor fornecido em cada um dos `case` seguintes. Se for encontrado um valor igual, todos os comandos após este **`case`** são executados, até encontrar um **`break`**. O **`default`**, no final do comando, é opcional. Os comandos após o **`default`** somente serão executados se o valor da variável não coincidir com nenhum dos valores correspondentes aos **`case`** anteriores.

Exemplo:

```
/*  prog15.c */
#include <stdio.h>

int main(void)
{
    int dia;

    printf("Informe o dia da semana (1-7): "); scanf("%d" , &dia);
    switch(dia){
        case 1:
            printf("Domingo\n");
            break;
        case 2:
            printf("Segunda-feira\n");
            break;
        case 3:
            printf("Terça-feira\n");
            break;
        case 4:
            printf("Quarta-feira\n");
            break;
        case 5:
            printf("Quinta-feira\n");
            break;
        case 6:
            printf("Sexta-feira\n");
            break;
        case 7:
            printf("Sábado\n");
            break;
        default:
            printf("Dia inválido\n");
    }
    fflush(stdin); getchar();
    return(0);
}
```

6. Funções

Além da função `main`, o programa pode conter outras funções, sendo que estas somente serão executadas se forem, direta ou indiretamente, chamadas pela função função principal. Uma função pode efetuar operações e pode chamar outras funções, sendo que estas podem fazer parte do próprio programa ou ser alguma das diversas funções pré-definidas na biblioteca padrão da linguagem C. Assim que a execução de uma função é concluída, ao atingir o **`}`** ou o comando **`return`**, a execução do programa retorna para o próximo comando após a chamada à esta função.

Sintaxe:

```
tipo_de_retorno nome_da_funcao(lista de parâmetros)
{
   declaração das variáveis locais

   corpo da função

   return(valor); ou return;
}
```

Toda a declaração de função em C deve seguir ao formato acima, onde aparecem os seguintes componentes, que devem ser declarados de acordo com a finalidade de cada função.

- **Tipo de retorno**: A função pode retornar (devolver) um valor para a função chamadora. O tipo de retorno da função indica qual o tipo de dados que será devolvido. Nos casos em que a função não retorna nenhum valor, funcionando como um procedimento, o tipo de retorno deve ser **`void`**.
- **Nome da função**: Indica o nome pelo qual o bloco de código correspondente à função será chamado.
- **Lista de parâmetros**: Os parâmetros de uma função são variáveis locais automaticamente inicializadas com os valores passados na posição correspondente no momento da sua chamada. Para cada um dos parâmetros deve ser declarado o seu tipo e nome. Se a função não receber nenhum parâmetro, deve-se declarar **`void`** neste local.
- **Declaração das variáveis locais**: Todas as variáveis locais devem ser declaradas dentro da função, antes de qualquer comando executável. Uma variável local somente pode ter seu valor acessado dentro da função em que foi declarada.
- **Corpo da função**: Conjunto de comandos que compõem a função. É onde é realizado o trabalho para o qual a função foi escrita.
- **Return**: O comando **`return`** finaliza a execução da função que está sendo executada. Se a função retornar algum valor (o tipo de retorno não é **`void`**) este comando é obrigatório. Se a função for **`void`**, pode-se simplesmente usar **`return;`** (sem nenhum valor), o que tem resultado idêntico a encontrar o fecha-chaves "`}`" que indica o final da função.

Esta forma de passagem de parâmetro, onde uma cópia do valor passado pela função chamadora é fornecido para a função chamada é denominado de *passagem de parâmetro por valor*.

Sempre que uma função é chamada acima do ponto onde ela foi criada, deve-se declarar o protótipo da função acima no início do programa. O protótipo corresponde à primeira linha da função (onde aparece o seu nome) finalizada por um ponto-e-vírgula.

```
/*  prog16.c
    Lê as duas notas de uma disciplina, calcula a média com
    uma casa decimal  e exibe o resultado.
*/
#include <stdio.h>

/* Prototipos */
float ler_nota(void);
float calcula_media(float p1, float p2);
void exibe_resultado(float med);
void pausa(void);

int main(void)
{
    float  n1, n2, med;
    printf("Primeira prova: "); n1=ler_nota();
    printf("Segunda prova : "); n2=ler_nota();
    med=calcula_media(n1, n2);
    exibe_resultado(med);
    pausa();
    return(0);
}

float ler_nota(void)
{
    float n;
    do{
        scanf("%f", &n);
        if(n<0.0 || n>10.0)
            printf("Inválido! redigite..:");
    }while(n<0.0 || n>10.0);
    return(n);
}

float calcula_media(float p1, float p2)
{
    float res;
    res = (p1 + p2) / 2.0;      /* mesmo peso na duas provas */
    res = ((int)(res*10))/10.0; /* trunca para a 1a casa decimal */
    return(res);
}

void exibe_resultado(float med)
{
    printf("Media %.1f: ", med);
    if(med < 3.0)
        printf("Aluno reprovado\n");
    else if(med >= 3.0 && med < 7.0)
        printf("Aluno em exame\n");
    else if(med >= 7.0)
        printf("Aluno aprovado\n");
}

void pausa(void)
{
    fflush(stdin);
    getchar();
}
```

7. Estruturas

Uma estrutura é o conjunto de variáveis agrupadas sob um nome único, sendo que estas variáveis podem ser de tipos de dados diferentes. A estrutura serve para organizar, de forma lógica, algum dado cujo valor é composto por mais de uma variável. Como exemplo pode-se citar uma data, que é composta por três valores, correspondentes ao dia, mês e ano.

7.1. Declaração de estruturas

As estruturas são definidas com a palavra reservada **`struct`**, normalmente no início do programa (antes da função **`main`**) ou em um arquivo separado (`.h`) incluído no programa que necessitar.

Sintaxe:

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

A etiqueta (*tag*) da estrutura deve ser um nome único e que será utilizado posteriormente para definição de variáveis. Entre o par de chaves (`{}`) deve-se declarar todas as variáveis que compõem a estrutura, que são chamadas de membros da estrutura. A declaração dos membros segue as regras para declaração de variáveis normais em C, exceto que não podem ser inicializados. Pode-se, inclusive, declarar vetores e outras esruturas como membros de estruturas. O tamanho total de uma estrutura, em bytes, é igual à soma do tamanho de seus membros.

A definição da estrutura, sozinha, não é o suficiente para que ela possa ser utilizada. Para isto é necessário a declaração de variáveis (quantas forem necessárias) para armazenamento dos valores. Para a declaração de variáveis uilizasse o tipo **`struct etiqueta_da_estrutura`**, seguido dos nomes das variáveis e, se for o caso, a inicialização da variável, com valores para cada um dos membros entre chaves e separados por vírgula, na ordem em que foram definidos. No corpo do programa, para acessar o valor de um membro da estrutura deve-se usar o operador **`.`** (ponto), unindo o nome da variável com o nome do membro.

Exemplo:

```
/*  prog17.c */
#include<stdio.h>

/* Definicao da estrutura */
struct data{
    int dia;
    int mes;
    int ano;
};

int main(void)
{
    struct data d1={30, 7, 2003}, d2;

    printf("Data-exemplo: %02d/%02d/%04d\n", d1.dia, d1.mes, d1.ano);
    printf("Informe outra data: ");
    scanf("%d/%d/%d", &d2.dia, &d2.mes, &d2.ano);
    if(d1.dia==d2.dia && d1.mes==d2.mes && d1.ano==d2.ano)
        printf("Datas iguais\n");
    else
        printf("Datas diferentes\n");

    fflush(stdin); getchar();
    return(0);
}
```

7.2. Vetores de estruturas

Vetores onde os elementos são estruturas podem ser declarados da mesma forma que outros tipos de vetores já estudados. A inicialização pode ser feita delimitando-se o conteúdo do vetor entre um par de chaves , assim como o conteúdo a ser atribuído a cada posição do vetor. O valor para cada um dos membros deve ser colocados na ordem de declaração, separados por vírgula. Para acessar um vetor de estruturas, deve-se colocar o índice, entre colchetes, logo à direita do nome do vetor, antes do ponto.

Exemplo:

```
/* prog18.c */
#include <stdio.h>
#define TAM_VET 4

struct data{
    int dia, mes, ano;
};

struct aluno{
    int matricula;
    char nome[30];
    struct data nascimento;
};

int main(void)
{
    struct aluno turma[TAM_VET] = {
        {9991, "Fulano", {10,5,1982}},
        {9992, "Cicrano", {23,8,1983}},
        {9993, "Beltrano", {14,1,1981}},
        {9994, "Individuo", {2,10,1983}}
    };
    int i, achou, m;
    char escolha;

    do{
        printf("Matricula: ");
        scanf("%d", &m);
        achou = 0;
        for(i=0; i<TAM_VET; i++){
            if(turma[i].matricula==m){
                printf("Nome: %s\n", turma[i].nome);
                printf("Nascimento: %02d/%02d/%04d\n",
                      turma[i].nascimento.dia,
                      turma[i].nascimento.mes,
                      turma[i].nascimento.ano);
                achou = 1;
                break;
            }
        }
        if(achou==0)
            printf("Nao Encontrado\n");
        printf("\nContinuar? [S/N]: ");
        scanf(" %c",&escolha);
    }while(escolha=='S' || escolha=='s');
    return(0);
}
```

8. Ponteiros

Normalmente, toda variável C é declarada para armazenar um valor numérico: variáveis **`int`** servem para armazenar valores inteiros, **`float`** para armazenar valores numéricos com precisão simples, **`char`** armazenam o código ASCII de um caracter, etc. Toda a vez que o nome de uma variável é inserido em um programa, está se fazendo referência ao seu valor, ou seja, o conteúdo daquela variável. Existe um tipo particular de variável em C que, ao invés de conter valores numéricos, armazena endereços de memória: São os **ponteiros**, também conhecidos por **apontadores**.

8.1. Declaração de ponteiros

Um ponteiro é uma variável declarada para armazenar o endereço de memória onde está armazenada outra variável. Declarações de ponteiros são precedidas do símbolo **`*`** (asterísco), como no exemplo abaixo, onde é declarada um ponteiro **`p`**, que pode conter qualquer endereço de memória onde está armazenado um valor inteiro (Lê-se: "`p` é um ponteiro para int").

Exemplo:

```
int *p;
```

Os ponteiros em C são tipados, isto é, devem ter um tipo declarado e somente podem apontar para variáveis do mesmo tipo, com exceção dos ponteiros para **`void`**, que podem apontar para variáveis de qualquer tipo, mas tem utilização limitada. Nenhum ponteiro pode ser usado antes de ser inicializado, isto é, enquanto não apontarem para um endereço válido sob o risco do programa ser abortado pelo sistema operacional por ter causado uma operação ilegal (acesso à endereco inválido de memória). A maneira mais simples de inicializar um ponteiro é fazê-lo apontar para uma variável existente.

8.2. Operadores específicos para ponteiros

Para trabalhar com ponteiros utiliza-se dois operadores específicos:

Operador	Descrição
`&`	Fornece o endereço de memória onde está armazenada uma variável. Lê-se *"o endereço de"*
`*`	Valor armazenado na variável referenciada por um ponteiro. Lê-se *"o valor apontado por"*

O operador **`&`**, quando colocado em frente ao nome de uma variável, obtem o endereço desta variável. Ao atribuir este endereço para um ponteiro, diz-se que o ponteiro está "apontando para" a variável. O operador **`*`** somente pode ser utilizado com ponteiros. Ele obtêm o valor da variável apontada pelo ponteiro, operação esta que é chamada de "referência indireta" .

Exemplo:

```
/* prog19.c */
#include <stdio.h>

int main(void)
{
   int v=25, *p;

   p = &v;                  /* p aponta para v */
   printf("%d %d\n", v, *p);
   *p = 50;                 /* atribuição por referencia indireta */
   printf("%d %d\n", v, *p);
   getchar();
   return(0);
}
```

8.2.1. Apontando para vetores

Por regra, sempre que em um programa aparece o nome de um vetor, sem o índice, isto é, sem os colchetes à sua direita, isto corresponde ao endereço do primeiro elemento daquele vetor. Assim a operação, comum em programas C, de fazer um ponteiro apontar para o primeiro elemento de um vetor pode ser simplificada de **`p=&vetor[0];`** para **`p=vetor;`**

Exemplo:

```
float vet[]={3.4, 17.0, 8.2}, *p;
p = vet;                                /* o mesmo que p = &vet[0]; */
printf("%f", *p);
```

Todo ponteiro que aponta para o primeiro elemento de um vetor pode ser utilizado como uma referência ao vetor apontado. Colocando-se um índice (entre colchetes) à direita do ponteiro, obter-se-á o valor correspondente na posição do vetor apontado. Deve-se chamar a atenção que o operador ***** não deve ser utilizado neste caso, apenas o colchete depois do ponteiro.

Exemplo:

```
/* prog20.c */
#include <stdio.h>

int main(void)
{
    char txt[100], *p;
    int i;

    printf("Digite algo: ");
    gets(txt);
    p=txt;                                /* p = &txt[0] */
    printf("Um caracter por linha:\n");
    for(i=0; p[i]!='\0'; i++)
        printf("%c\n", p[i]);             /* p[i] corresponde a txt[i] */

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

8.3. Operações aritméticas com ponteiros

Pode-se somar ou subtrair um valor de um ponteiro, mas estas operações normalmente só tem sentido com ponteiros que apontam para elementos de um vetor. Incrementar um destes ponteiros fará com que ele aponte para o próximo elemento dentro do vetor referenciado. Decrementá-lo faz com que ele aponte para o elemento anterior.

Exemplo:

```
/* prog21.c */
#include <stdio.h>

int main(void)
{
    char txt[100], *p;

    printf("Digite algo: ");
    gets(txt);
    printf("Um caracter por linha:\n");
    for(p=txt; *p!='\0'; p++)
        printf("%c\n", *p);

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

8.4. Passagem de parâmetros por referência

Se no lugar de passar uma cópia do valor de uma variável como argumento de uma função, for passado o seu endereço de memória, qualquer alteração feita usando referência indireta irá modificar o valor da variável utilizada na chamada da função. Isto é *passagem de parâmetro por referência*. Para isto, a variável que recebe o parâmetro, na função chamada, deve ser declarado como um ponteiro. A mesma função, se receber mais que um argumento, pode combinar passagem de parâmetro por referência e por valor.

Exemplo:

```
/* prog22.c */
# include <stdio.h>

void troca(int *p1, int *p2);

int main(void)
{
   int a, b;

   printf("A: "); scanf("%d", &a);
   printf("B: "); scanf("%d", &b);
   printf("Antes>  A: %d   B: %d\n", a, b);
   troca(&a, &b);
   printf("Depois> A: %d   B: %d\n", a, b);
   fflush(stdin); getchar();
   return(0);
}

void troca(int *p1, int *p2)
{
    int temp;

    temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

8.5. Vetores como argumento de funções

Sempre que vetores forem passados como parâmetros para função, obrigatóriamente eles devem ser passados por referência. Passa-se, então o endereço do primeiro elemento do vetor para um parâmetro declarado como um ponteiro para o tipo adequado.

Exemplo com string:[1]

```
/* prog23.c */
#include <stdio.h>

void substitui(char *str, int atual, int novo)
{
    int i;

    for(i=0;str[i]!='\0';i++)
        if(str[i]==atual)
            str[i]=novo;
}
```

1 Este exemplo continua na próxima página

```
int main(void)
{
    char txt[100]="aula";

    printf("Original: [%s]\n", txt);
    substitui(txt,'a','o');
    printf("Resultado: [%s]\n", txt);
    getchar();
    return(0);
}
```

Exemplo com vetor `float`:

```
/* prog24.c */
#include<stdio.h>
#define TAMVET 5

int fposmax(float *vetor, int n);

int main(void)
{
    float vet[TAMVET];
    int i, pos;

    printf("Digite %d valores\n", TAMVET);
    for(i=0; i<TAMVET; i++){
        printf("%d> ", i);
        scanf("%f", &vet[i]);
    }
    pos=fposmax(vet, TAMVET);
    printf("O maior valor é %.2f e está no indice %d\n", vet[pos], pos);
    fflush(stdin); getchar();
    return(0);
}

/* Função fposmax - Encontra a posição do maior valor em um vetor float
   Parâmetros:
     vetor: endereço do primeiro elemento do vetor a ser pesquisado
     n: tamanho do vetor (número de elementos)
   Retorno: Índice do maior valor.
            Se este valor está repetido, retorna o primeiro índice.
*/
int fposmax(float *vetor, int n)
{
   int posmaior=0, i;

   for(i=0; i<n; i++){
        if(vetor[i] > vetor[posmaior]){
            posmaior = i;
        }
    }
    return(posmaior);
}
```

8.6. Alocação dinâmica de memória

Por alocação de memória entende-se a ação de reservar uma área da memória do computador para o armazenamento de dados. Todas as variáveis que foram utilizadas até este momento, incluindo as de tipos básicos, os arranjos e as estruturas, foram alocadas *estaticamente*, isto é, o seu tamanho e duração são definidos no momento em que o programa é compilado (em *tempo de compilação ou compile time*). Isto, porém, não é adequado para todas as aplicações. Em casos onde em que o tamanho necessário para um arranjo só é conhecido no momento em que o programa estiver sendo executado (em *tempo de execução ou run time*) ou quando se quer criar ou destruir variáveis em qualquer parte do programa precisa-se empregar recursos de *alocação dinâmica de memória*. As funções da biblioteca padrão C para isto estão definidas em `<stdlib.h>`:

• **`void *malloc(int tamanho);`**

Descrição:

Aloca dinamicamente uma área de memória com o tamanho solicitado (em bytes) .

Parâmetros:

- **`tamanho`**: Quantidade de bytes a serem reservados. Deve-se chamar a atenção de que, por ser definido em bytes, na alocação de arranjos o tamanho não é igual ao número de elementos. Deve-se utilizar a fórmula **`num_elementos*sizeof(tipo)`**.

Valor de retorno:

Retorna o endereço do início da memória alocada, ou retorna **`NULL`** se não existir memória disponível suficiente. O endereço retornado deve ser atribuído para um ponteiro de tipo adequado.

• **`void *realloc(void *mem, int tamanho);`**

Descrição:

Altera o tamanho da área alocada anteriormente por `malloc`.

Parâmetros:

- **`mem`**: Endereço da memória a ser realocada. Deve-se passar nesta posição o ponteiro que anteriormente recebeu o retorno da função **`malloc`**.
- **`tamanho`**: Novo tamanho, em bytes, da área alocada. Este valor pode ser maior ou menor que o originalmente alocado.

Valor de retorno:

Se a área de memória não puder ser realocada retorna **`NULL`**, senão retorna o endereço da área alocada, que deve ser atribuído novamente ao ponteiro.

• **`void free(void *mem);`**

Descrição:

Libera a área de memória alocada previamente por **`malloc`**. Toda memória alocada dinamicamente deve ser liberada, sob pena de perder-se parte da memória disponível do computador (*heap corruption*).

Parâmetro:

- **`mem`**: Endereço da memória a ser liberada. Deve-se passar por parâmetro o ponteiro que anteriormente recebeu o retorno da função **`malloc`**.

Valor de retorno:

Esta função não retorna nenhum valor.

Exemplo:

```
/*  prog25.c
    Cria um vetor dinâmico de tamanho 'n',
    lê valores e calcula soma e media.
*/
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int *vet, n, i, soma=0;
    float media;

    printf("Numero de elementos do vetor: ");
    scanf("%d", &n);
    vet = malloc(n*sizeof(int));

    if(vet==NULL){
        printf("Sem Memoria!\n");
        return(1);
    }

    for(i=0;i<n;i++){
        printf("[%d]: ", i);
        scanf("%d", &vet[i]);
        soma += vet[i];
    }

    media=(float)soma/n;
    printf("Soma: %d\nMedia: %.2f\n", soma, media);
    free(vet);
    fflush(stdin); getchar();
    return(0);
}
```

8.7. Ponteiros para estruturas

Ponteiros que apontem para variáveis de estruturas podem ser declarados e inicializados da mesma forma que os ponteiros normais. A única particularidade deste tipo de ponteiro é que, para acessar indiretamente o valor armazenado no membro de uma estrutura através de um ponteiro usa-se o operador **->** (lê-se *"seta"*) e não o * (asterísco) utilizado nos demais tipos de ponteiros.

Exemplo:

```
struct data hoje={17, 7, 2002}, *ptr;
ptr = &hoje;
printf("Ano: %d\n", ptr->ano);
```

Os ponteiros para estrutura são utilizados, principalmente, na chamada de funções. Isto por que as estruturas não podem ser passadas por valor, apenas por referência. Passa-se, então, o endereço da variável para um ponteiro. Não é permitido também retornar uma estrutura em uma função mas pode-se retornar o endereço dela.

Exemplo:

```
/* prog26.c */
#include<stdio.h>

struct data{
    int dia, mes, ano;
};

/* Recebe uma data como parâmetro.
   Retorna 1 se é válida ou 0 se não é válida.
*/
int datavalida(struct data *pd)
{
    int dias[12]={31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};


    if(pd->ano < 0)
        return(0);
    if(pd->mes < 1 ||pd->mes > 12)
        return(0);
    if((pd->ano%4==0 && pd->ano%100!=0) || pd->ano%400==0)
        dias[1]=29;
    if(pd->dia < 1 || pd->dia > dias[pd->mes-1])
        return(0);

    return(1);
}

int main(void)
{
    struct data dt;
    int flag;

    printf("Informe uma data: ");
    do{
        scanf("%d/%d/%d", &dt.dia, &dt.mes, &dt.ano);
        flag=datavalida(&dt);
        if(flag==0)
            printf("Data invalida! Redigite...:");
    }while(flag==0);

   fflush(stdin); getchar();
   return(0);
}
```

9. Resumo da biblioteca padrão

9.1. Funções de manipulação de strings

A biblioteca `<string.h>` fornece um conjunto de funções para manipulação de strings. Estas funções são particularmente úteis, devido ao fato de que em C toda string é um vetor de caracteres e, devido a isto, herda uma série de limitações pertinentes aos vetores. Principalmente pelo fato da linguagem C não suportar operações com vetores utilizando-se diretamente os operadores estudados (aritméticos e relacionais). Desta forma, todo o programa que manipular strings deve tomar cuidado, principalmente, com as seguintes limitações:

1. Strings não podem ser atribuídas com o operador de atribuição (=), embora possam ser inicializadas;
2. Strings não podem ser comparadas com os operadores relacionais (`==, !=, >, >=, <, <=`).

Na seqüência será apresentado um resumo do conjunto de funções disponibilizados por `string.h`:

• `char *strcpy(char *destino, char *origem);`

Descrição:

Copia o conteúdo de uma string para outra. Deve ser empregado para atribuição de strings, no lugar do operador normal de atribuição (=).

Parâmetros:

- **`destino`**: String que irá receber o conteúdo.
- **`origem:`** String cujo conteúdo será copiado.

Valor de retorno:

A função retorna o endereço do primeiro caracter da string **`destino`**.

• `char *strncpy(char *destino, char *origem, int n);`

Descrição:

Copia no máximo **`n`** caracteres de uma string para a outra. Não coloca o `'\0'` no final de destino., a não ser que tenha atingido o final da string **`origem`**.

Parâmetros:

- **`destino`**: String que irá receber o conteúdo.
- **`origem:`** String cujo conteúdo será copiado.
- **`n:`** Número máximo de caracteres a serem copiados.

Valor de retorno:

A função retorna o endereço do primeiro caracter da string **`destino`**.

• `char *strcat(char *destino, char *origem);`

Descrição:

Concatena strings, isto é, copia o conteúdo da string **`origem`** ao final da string **`destino`**. O conteúdo anterior de **`destino`** é preservado.

Parâmetros:

- **`destino`**: String que irá receber, no seu final, o conteúdo. Ela deve ter tamanho suficiente para armazenar o conteúdo atual mais o novo.
- **`origem`**: String cujo conteúdo será acrescentado ao final da outra.

Valor de retorno:

A função retorna o endereço do primeiro caracter da string **`destino`**.

- **`int strcmp(char *s1, char *s2);`**

Descrição:

Compara o conteúdo de duas strings. Esta função deve ser utilizada em substituição aos operadores relacionais no caso de uso com strings.

Parâmetros:

- **`s1`** e **`s2:`** As duas strings a serem comparadas.

Valor de retorno:

A função retorna 0 (zero) se o conteúdo de ambas as strings são iguais. Retorna algum valor maior que 0 se o conteúdo de s1 é maior que s2 e um valor menor que 0 se o conteúdo de s1 é menor que s2. Quando se trata de comparação de strings, maior e menor não se refere ao tamanho, mas à posição, quando ordenadas de forma ascendente.

- **`int strncmp(char *s1, char *s2, int n);`**

Descrição:

Compara apenas um trecho do início de duas strings com tamanho especificado.

Parâmetros:

- **`s1`** e **`s2:`** As duas strings a serem comparadas.
- **`n`**: Número de caracteres a serem comparados.

Valor de retorno:

O mesmo da função **`strcmp`** (ver acima).

- **`int strcasecmp(char *s1, char *s2);`**

Descrição:

Compara o conteúdo de duas strings sem diferenciar a caixa (letras maiúsculas ou minúsculas).

Parâmetros:

- **`s1`** e **`s2:`** As duas strings a serem comparadas.

Valor de retorno:

O mesmo da função **`strcmp`** (ver acima).

- **`int strlen(char *str);`**

Descrição:

Conta o número de caracteres armazenados em uma string, antes do `'\0'`.

Parâmetros:

- **`str`**: A string que terá seu tamnho calculado.

Valor de retorno:

Retorna o número de caracteres da string.

Exemplo:

```
/* prog27.c */
#include<stdio.h>
#include<string.h>

void inserestr(char *destino, char *origem, char *outra, int pos);

int main(void)
{
    char s1[100], s2[100], s3[100];
    int pos;

    printf("String: ");
    gets(s1);
    printf("Inserir: ");
    gets(s2);
    printf("Posição: ");
    fflush(stdin);
    scanf("%d", &pos);
    inserestr(s3, s1, s2, pos);
    printf("%s\n", s3);
    fflush(stdin); getchar();
    return(0);
}

/*
     Copia o conteúdo de origem para destino,
     inserindo outra na posicão indicada.
     Se a posição não for válida, apenas copia origem, sem inserir
*/
void inserestr(char *destino, char *origem, char *outra, int pos)
{
    if(pos<0 || pos > strlen(origem)){
        strcpy(destino, origem);
        return;
    }

    strncpy(destino, origem, pos);
    destino[pos]='\0';
    strcat(destino, outra);
    strcat(destino, &origem[pos]);
}
```

9.2. Funções de tratamento de caracteres

As funções de <ctype.h> são utilizadas para manipulação de caracteres, sendo utilizadas para dois tipos de operação: testar se um caracter é de uma determinada classe ou converter a caixa (*case*) de um caracter. Os nomes das funções do primeiro tipo começam com "**is**", enquanto que as demais iniciam com "**to**". Sempre os parâmetros destas funções devem ser um único caracter, nunca uma string (pode ser uma posição de string).

9.2.1. Funções de teste de classe

- **int isalnum(int c);**

Descrição:
Testa se o caracter passado por parâmetro é uma letra ou um dígito numérico.
Parâmetros:
- **c**: O caracter a ser testado.

Valor de retorno:
Retorna verdadeiro (um valor diferente de 0) se a condição foi satisfeita, ou falso (0) em outro caso.

- **int isalpha(int c);**

Descrição:
Testa se o caracter passado por parâmetro é uma letra.
Parâmetros:
- **c**: O caracter a ser testado.

Valor de retorno:
Retorna verdadeiro (um valor diferente de 0) se a condição foi satisfeita, ou falso (0) em outro caso.

- **int isdigit(int c);**

Descrição:
Testa se o caracter passado por parâmetro é um dígito numérico.
Parâmetros:
- **c**: O caracter a ser testado.

Valor de retorno:
Retorna verdadeiro (um valor diferente de 0) se a condição foi satisfeita, ou falso (0) em outro caso.

- **int islower(int c);**

Descrição:
Testa se o caracter passado por parâmetro é uma letra minúscula.
Parâmetros:
- **c**: O caracter a ser testado.

Valor de retorno:
Retorna verdadeiro (um valor diferente de 0) se a condição foi satisfeita, ou falso (0) em outro caso.

- **int isupper(int c);**

Descrição:
Testa se o caracter passado por parâmetro é uma letra maiúscula.
Parâmetros:
- **c**: O caracter a ser testado.

Valor de retorno:
Retorna verdadeiro (um valor diferente de 0) se a condição foi satisfeita, ou falso (0) em outro caso.

- **int isspace(int c);**

Descrição:
Testa se o caracter passado por parâmetro é um espaço em branco. O caracter correspondente à barra de espaços, quebra de linha e tabulações são considerados brancos.
Parâmetros:
- **c**: O caracter a ser testado.

Valor de retorno:
Retorna verdadeiro (um valor diferente de 0) se a condição foi satisfeita, ou falso (0) em outro caso.

9.2.2. Funções de conversão de caracter

- **`int tolower(int c);`**

Descrição:
Converte uma letra para o formato minúsculo
Parâmetros:
- **`c`**: O caracter a ser convertido.

Valor de retorno:
Se o caracter for uma letra, retorna-o convertido para minúsculo, senão retorna o caracter sem alteração.

- **`int toupper(int c);`**

Descrição:
Converte uma letra para o formato maiúsculo
Parâmetros:
- **`c`**: O caracter a ser convertido.

Valor de retorno:
Se o caracter for uma letra, retorna-o convertido para maiúsculo, senão retorna o caracter sem alteração

Exemplo:

```
/* prog28.c */
#include <stdio.h>
#include <ctype.h>

void nomeproprio(char *destino, char *origem);

int main(void)
{
    char s1[100], s2[100];

    printf("Digite um Nome: ");
    gets(s1);
    nomeproprio(s2, s1);
    printf("Original: [%s]\nModificado: [%s]\n", s1, s2);
    getchar();
    return(0);
}

/* Função nomeproprio: copia o conteúdo da string origem para destino
    convertendo a primeira letra de cada palavra para maiúsculo
    e as demais para minúsculo
*/
void nomeproprio(char *destino, char *origem)
{
    int i;
    char anterior=' ';

    for(i=0; origem[i]!='\0'; i++){
        if(isspace(anterior))
            destino[i] = toupper(origem[i]);
        else
            destino[i] = tolower(origem[i]);

        anterior = origem[i];
    }
    destino[i] = '\0';
}
```

9.3. Funções diversas

O arquivo `<stdlib.h>` ("**st**an**d**ard **lib**rary", ou "biblioteca padrão") contêm um conjunto de funções variadas, muitas delas sem relação entre sí, ao contrario dos outros *includes*. Além das funções de alocação dinâmica vistas anteriormente (pg. 27), pode-se destacar as seguintes funções:

- **`void exit(int codigo);`**

Descrição:

Esta função finaliza a execução do programa, podendo ser chamada a partir de qualquer função do programa, ou em substituição ao **`return`**, na função **`main`**.

Parâmetros:

- **`codigo`**: Valor a ser retornado pelo programa. Por padrão deve ser 0 (zero) no caso de fim normal, ou um valor maior que 0, caso o programa esteja sendo finalizado por alguma situação anormal.

Valor de retorno:

Esta função não retorna.

- **`int atoi(char *str);`**

Descrição:

Converte uma string para um valor numérico inteiro.

Parâmetros:

- **`str`**: String a ser convertida.

Valor de retorno:

Retorna um número inteiro correspondente ao valor armazenado na string.

- **`double atof(char *str);`**

Descrição:

Converte uma string para um valor numérico em ponto flutuante.

Parâmetros:

- **`str`**: String a ser convertida.

Valor de retorno:

Retorna um valor **`double`** correspondente ao conteúdo armazenado na string.

10. Argumentos de linha de comando

Argumentos de linha de comando são valores fornecidos pelo usuário para o programa no momento em que este é invocado, através do *Shell Unix*, do *Prompt do MS-DOS*, da janela *Executar*, no Windows, entre outras formas. Conciste em todos os termos (palavras, opções, etc.) que forem digitadas com o nome do programa e normalmente são opcionais, mas se forem utilizados servem principalmente para fornecer opções (chaves) que irão alterar o comportamento padrão do programa e indicar nomes de arquivos a serem processados. Cada um dos argumentos é separado, na linha de comando, por um ou mais espaços em branco e sempre são disponibilizados ao programa na forma de strings. Tomemos um exemplo onde o usuário digita, no *shell*, o seguinte comando

```
$ gcc teste.c -o teste
```

ou o equivalente, no prompt do MS-DOS:

```
c:\> gcc teste.c -o teste
```

Nestes casos, a linha de comando é formada por quatro strings: `"gcc"` (o nome do programa a ser executado), `"teste.c"`, `"-o"` e `"teste"`. Todo o programa escrito em libguagem C, para estar preparado a receber argumentos pela linha de comando, deve ter a declaração da função **main** modificada para o seguinte formato: `int main(int argc, char **argv)`

Os seguintes parâmetros são fornecidos pelo sistema operacional para o programa que está sendo executado:

- **`argc`**: Número de argumentos fornecidos na linha de comando, contando o próprio nome do programa. Portanto o valor de `argc` sempre será maior ou igual a 1 (um).
- **`argv`**: É um vetor de strings (na forma de ponteiro para ponteiros para `char`) onde cada posição aponta para um dos argumentos passados. `argv[0]` é sempre o nome do programa, `argv[1]` o segundo argumento fornecido, e assim por diante. O tamanho deste vetor (número de elementos) é igual ao valor de `argc`.

Exemplo[2]:

```
/* prog29.c
   Soma o conjunto (variável) de valores inteiros positivos
   digitados na linha de comando
*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int isnumeric(char *str);

int main(int argc, char **argv)
{
    int i, total=0;

    for(i=1; i<argc; i++){
        if(isnumeric(argv[i]))
            total+=atoi(argv[i]);
        else{
            printf("Valor \"%s\" inválido.\n", argv[i]);
            exit(1);
        }
    }
    printf("Total: %d\n", total);
    return(0);
}
```

2 Continua na próxima página...

```
/* Testa se a string contem apenas digitos numéricos */
int isnumeric(char *str)
{
    int i;
    for(i=0; str[i]!='\0'; i++)
        if(!isdigit(str[i]))
            return(0);
    return(1);
}
```

11. Funções de entrada e saída em arquivos

A biblioteca `<stdio.h>` provê um vasto conjunto de funções e macros para entrada e saída (E/S), além daquelas já estudadas (pg. 8) e até aqui utilizadas para ler e escrever nos dispositivos padrão (normalmente vídeo e teclado). Estas novas funções são úteis para E/S em arquivos, normalmente armazenados em meio magnético, embora, por definição, no mundo Unix, qualquer dispositivo conectado ao sistema pode ser considerado um arquivo (inclusive teclado, modem, impressora, etc.).

Todo arquivo precisa ser aberto para que o seu conteúdo esteja disponível ao programa. A ação de abrir o arquivo envolve reservar áreas de memória para armazenamento temporário de dados necessários à transferência e a solicitação do acesso ao sistema operacional. Após a abertura, se esta teve sucesso, o programa pode utilizar as funções adequadas para ler ou escrever dados no arquivo aberto. Eventualmente a abertura de um arquivo pode falhar, como nos casos em que o arquivo a ser lido não existe, o usuário não tem permissão de acesso ao arquivo ou diretório, entre outros. Finalmente, após os dados terem sido processados pelo programa e quando este não necessitar mais acessar o conteúdo do arquivo, este deve ser fechado.

Todo o programa C que necessitar abrir arquivos deverá declarar, para cada arquivo aberto, uma variável do tipo **`FILE *`**. Esta variável será associada com o nome do arquivo no momento da abertura e todo o acesso posterior, através das funções adequadas, fará uso desta variável. A seguir, serão apresentadas funções utilizadas para manipulação de arquivos.

- **`FILE *fopen(char *nome,char *modo)`**

Descrição:

Abre um arquivo, tornando-o disponível a ser acessado pelo programa

Parâmetros:

- **`nome`**: String contendo o nome do arquivo a ser aberto.
- **`modo`**: Modo de abertura do arquivo. Indica qual tipo de acesso ao arquivo está sendo solicitado para o sistema operacional. Deve ser uma string contendo uma combinação válida dos caracteres apresentados na tabela abaixo.

Caracter	Descrição
`"r"`	Abre o arquivo somente para leitura
`"w"`	Cria um novo arquivo para escrita. Se já existir um arquivo com o mesmo nome, ele é eliminado e recriado vazio
`"a"`	Abre um arquivo para escrita no final do arquivo
`"+"`	Em conjunto com uma opção anterior permite acesso de leitura e escrita
`"b"`	Arquivo binário*
`"t"`	Arquivo texto*

*: As opções `"b"` e `"t"` somente são úteis no DOS/Windows: elas são ignoradas no Unix

Valor de retorno:

Se o arquivo foi aberto, retorna um endereço que deve ser atribuído para uma variável de tipo **`FILE *`** para uso posterior com as outras funções. Retorna **`NULL`** em caso de erro.

- **`int fclose(FILE *arquivo)`**

Descrição:

Fecha um arquivo aberto, tornando-o indisponível para o programa.

Parâmetros:

- **`arquivo`**: A variável que recebeu o valor de retorno da função **`fopen`**, correspondente ao arquivo a ser fechado.

Valor de retorno:

A função retorna 0 (zero) se o arquivo pode ser fechado ou **`EOF`** se algum erro ocorreu.

Alguns arquivos, associados à dispositivos muito utilizados, são definidos e abertos automaticamente pelo C, podendo ser utilizados por qualquer programa sem a necessidade de declaração de variável e chamada à função **`fopen:`**

Arquivo	Descrição
`stdin`	Dispositivo de entrada padrão: normalmente o teclado, mas podem ser redirecionado
`stdout`	Dispositivo de saída padrão: normalmente o vídeo, mas podem ser redirecionado
`stderr`	Dispositivo de saída de erros: o vídeo

- **`int fprintf(FILE *arquivo,char *formato,...)`**

Descrição:

Similar à função **`printf`**, mas permite que seja especificado o arquivo de saída (**`printf`** sempre imprime em **`stdout`**).

Parâmetros:

- **`arquivo`**: Em qual arquivo será escrito o conteúdo. È muito comum ser utilizado com **`stderr`** para apresentar mensagens de erros e advertências ao usuário.
- **`formato`**: A string de formato idêntica à da função **`printf`**.
- **`...`**: conjunto variável de dados a serem escritos.

Valor de retorno:

Assim como **`printf`**, esta função retorna o número de caracteres que foram escritos.

11.1. Funções de acesso sequencial

O acesso sequencial (ou serial) em arquivos caracteriza-se pelo fato de que cada operação de leitura ou escrita acessa a posição imediatamente seguinte à operação anterior. Cada arquivo possui associado um *cursor* que indica qual a posição onde será feita o próximo acesso. Quando um arquivo é aberto este cursor está na sua posição inicial (exceto se ele foi aberto com o modo de abertura `"a"`). Após cada operação de escrita ou leitura, este cursor avança automaticamente para a próxima posição no arquivo. O arquivo possui uma marca de final de arquivo (**`EOF`**) que pode ser utilizada para testar se o final foi atingido.

11.1.1. Leitura e escrita de caracteres

As seguintes funções podem ser utilizadas para ler/escrever um caracter por vez de/para um arquivo aberto. Apesar dela efetuarem operações com caracteres, os valres retornados por estas funções, se forem armazenados em variáveis, estas devem ser declaradas com o tipo **`int`**.

- **`int fgetc(FILE *arquivo)`**

Descrição:

Lê o próximo caracter de um arquivo aberto.

Parâmetros:

- **`arquivo`**: A variável que recebeu o valor de retorno da função **`fopen`**, correspondente ao arquivo que será lido.

Valor de retorno:

Retorna o caracter lido, em caso de leitura bem sucedida, ou **`EOF`**, se o final do arquivo foi alcançado.

- **`int fputc(int c, FILE *arquivo)`**

Descrição:

Escreve um caracter em um arquivo aberto.

Parâmetros:

- **`c`**: O caracter a ser gravado no arquivo.
- **`arquivo`**: A variável que recebeu o valor de retorno da função **`fopen`**, correspondente ao arquivo que será escrito.

Valor de retorno:

Retorna o próprio caracter , se ele foi escrito com sucesso , ou **`EOF`**, em caso de erro.

Exemplo:

```
/* prog30.c
   Recebe o nome de dois arquivos na linha de comando e
   copia o conteúdo do primeiro arquivo para o segundo
*/
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    FILE *origem,*destino;
    int c;

    if(argc!=3){
        fprintf(stderr, "Numero de argumentos incorreto\n");
        exit(1);
    }
    origem=fopen(argv[1],"rb");
    if(origem==NULL){
        fprintf(stderr, "Nao consegui abrir %s\n",argv[1]);
        exit(1);
    }
    destino=fopen(argv[2],"wb");
    if(destino==NULL){
        fprintf(stderr, "Nao consegui criar %s\n",argv[2]);
        exit(1);
    }
    do{
        c=fgetc(origem);
        if(c==EOF) break;
        fputc(c,destino);
    }while(1);
    fclose(origem);
    fclose(destino);
    return(0);
}
```

11.1.2. Leitura e escrita de strings

- **`char *fgets(char *str, int n, FILE *arquivo)`**

Descrição:

Lê a próxima linha de texto em um arquivo, armazenando-a em uma string.

Parâmetros:

- **`str`**: Variável string que irá receber o conteúdo da linha lida (inclusive o `'\n'`).
- **`n`**: Limita o número máximo de caracteres a serem lidos em uma linha em `n-1` caracteres. Normalmente este valor é igual ao tamanho da string, ou seja, **`sizeof(str)`**. Se a linha armazenada no arquivo for maior que este tamanho, é lida apenas a sua parte inicial. O restante será lido na próxima chamada à função **`fgets`**.
- **`arquivo`**: A variável que recebeu o valor de retorno da função **`fopen`**, correspondente ao arquivo que será lido.

Valor de retorno:

Retorna o endereço do primeiro caracter da string lida, em caso de sucesso, ou **`NULL`**, se o final de arquivo foi alcançado.

Devido à problemas de segurança da função **`gets`**, é mais adequado utilizar **`fgets`**, lendo do arquivo **`stdin`**, no caso de leitura de strings na entrada-padrão: **`fgets(str, sizeof(str), stdin);`**

- **`int fputs(char *str,FILE *arquivo)`**

Descrição:

Escreve uma string em um arquivo (exceto o `'\0'`).

Parâmetros:

- **`str`**: A string que possui o conteúdo a ser escrito.
- **`arquivo`**: A variável que recebeu o valor de retorno da função **`fopen`**, correspondente ao arquivo que será escrito.

Valor de retorno:

Retorna **`EOF`**, em caso de erro, ou algum valor positivo, se a escrita foi bem sucedida.

```
/* prog31.c
   Recebe o nome de um arquivo-texto na linha de comando e
   exibe o seu conteúdo.
*/
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    FILE *arq;
    char linha[255];

    if(argc!=2){
        fprintf(stderr, "Numero de argumentos incorreto\n");
        exit(1);
    }
    arq=fopen(argv[1],"rt");
    if(arq==NULL){
        fprintf(stderr, "Nao consegui abrir %s\n",argv[1]);
        exit(1);
    }

    while(fgets(linha,sizeof(linha),arq)!=NULL)
        fputs(linha, stdout);

    fclose(arq);
    return(0);
}
```

11.2. Funções de acesso aleatório

Este método somente pode ser utilizado quando os registros que compõem um arquivo possuem um tamanho fixo e determinado. Cada registro possui associado um número inteiro que indica a sua posição relativa ao início do arquivo. Cada vez que se lê/escreve *n* bytes no arquivo este cursor é incrementado em *n*. A diferença, em comparação ao acesso sequencial, é que este cursor pode ser reposicionado pelo programador, através de funções apropriadas, a fim de localizar o dado a ser lido. Normalmente (mas não obrigatoriamente) estas funções são utilizadas para a leitura e escrita de variáveis de estruturas (`struct`) em um arquivo.

• int fread(void *endereco, int tamanho, int num, FILE *arquivo)

Descrição:

Lê um conjunto de bytes de um arquivo e armazena-os na posição de memória indicada no primeiro parâmetro.

Parâmetros:

- **endereco**: Endereço da área de memória onde serão armazenados os dados lidos do arquivo.
- **tamanho**: Tamanho, em bytes, da variável a ser lida.
- **num**: Número de variáveis a serem lidas em um único acesso. Geralmente 1 (um).
- **arquivo**: A variável que recebeu o valor de retorno da função **fopen**, correspondente ao arquivo que será lido.

Valor de retorno:

A função retorna o número de variáveis realmente lidas e avança o cursor de arquivo *tamanho*num* bytes.

• int fwrite(void *buffer, int tamanho, int num, FILE *arquivo)

Descrição:

Escreve um conjunto de bytes em um arquivo. Se o cursor estiver apontando para uma área já existente do arquivo, então os novos dados irão sobrescrever os anteriores. Se estiver apontando para o final do arquivo, então o tamanho do arquivo será aumentado e os os novos dados serão anexados.

Parâmetros:

- **endereco**: Endereço da área de memória onde estão os dados a serem escrito no arquivo.
- **tamanho**: Tamanho, em bytes, da variável a ser escrita.
- **num**: Número de variáveis a serem gravadas na mesma operação. Geralmente 1.
- **arquivo**: A variável que recebeu o valor de retorno da função **fopen**, correspondente ao arquivo que será escrito.

Valor de retorno:

A função retorna o número de variáveis realmente gravados e avança o cursor *tamanho*num* bytes.

• int fseek(FILE *arquivo, int deslocamento, int onde)

Descrição:

Altera a posição do cursor de um arquivo, indicando onde será feito o próximo acesso ao arquivo.

Parâmetros:

- **arquivo**: A variável que recebeu o valor de retorno da função **fopen**, correspondente ao arquivo cujo cursor será reposicionado.
- **deslocamento**: Quantidade de bytes que o cursor será ser movimentado. Este valor depende do parâmetro a seguir:
- **onde**: Indica uma da posições possíveis, relativas a qual a movimentação será feita. Deve-se utilizar uma das seguintes macros:

Valor	Descrição
`SEEK_SET`	Posiciona a partir do início do arquivo
`SEEK_CUR`	Relativo à posição atual
`SEEK_END`	Retrocede do final do arquivo

Valor de retorno:

Retorna 0 se OK, ou **EOF**, em caso de erro.

• **void rewind(FILE *arquivo)**

Descrição:

Posiciona o cursor no início do arquivo. É identico à **fseek(arquivo,0,SEEK_SET);**

Parâmetros:

- **arquivo**: A variável que recebeu o valor de retorno da função **fopen**, correspondente ao arquivo cujo cursor será reposicionado.

Valor de retorno:

Esta função não retorna valor.

• **int ftell(FILE *arquivo)**

Descrição:

Obtêm a posição atual do cursor de arquivo, isto é, em qual posição será feita a próxima operação de escrita ou leitura.

Parâmetros:

- **arquivo**: A variável que recebeu o valor de retorno da função **fopen**, correspondente ao arquivo cuja posição do cursor será obtida.

Valor de retorno:

Retorna a posição atual do cursor, na forma de um número inteiro positivo, ou **EOF** no caso de erro.

• **int feof(FILE *arquivo)**

Descrição:

Testa se o final do arquivo foi atingido.

Parâmetros:

- **arquivo**: A variável que recebeu o valor de retorno da função **fopen**, correspondente ao arquivo cujo final será testado.

Valor de retorno:

Retorna um valor diferente de zero (verdadeiro) se o programa tentou ultrapassar o final do arquivo, ou zero (falso) caso contrário.

Exemplos:

```
/* temperaturas.h */

#define NOMEARQUIVO "temperaturas.dat"

struct data{
      int dia, mes, ano;
};

struct temperatura{
      struct data quando;
      float minima, maxima;
};
```

```
/* prog32.c */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "temperaturas.h"

int main(void)
{
     FILE *arq;
    struct data nova;
    struct temperatura td;
     int achou;
    char escolha;

     /*Tenta abrir para leitura/escrita*/
     arq=fopen(NOMEARQUIVO,"r+b");
     if(arq==NULL){ /* Se não abriu, tenta criar novo */
          arq = fopen(NOMEARQUIVO,"w+b");
          if(arq==NULL){
               fprintf(stderr,"Nao consegui criar %s\n", NOMEARQUIVO);
               exit(1);
          }
     }
     do{
          printf("\nData : ");
          scanf("%d/%d/%d", &nova.dia, &nova.mes, &nova.ano);
          /* Posiciona no inicio do arquivo */
          rewind(arq);
          achou = 0;
          /* Lê até o final ,procurando pela data */
          while(fread(&td,sizeof(td),1,arq)==1 && !feof(arq))
               if(nova.dia==td.quando.dia &&
                  nova.mes==td.quando.mes &&
                  nova.ano==td.quando.ano){
                    printf("Data ja cadastrada!\n");
                    achou = 1;
                    break;
          }
          if(achou==0){
               td.quando.dia = nova.dia;
               td.quando.mes = nova.mes;
               td.quando.ano = nova.ano;
               printf("Temperatura Minima: ");
               scanf("%f", &td.minima);
               printf("Temperatura Maxima: ");
               scanf("%f", &td.maxima);
               /* Posiciona o cursor no final do arquivo */
               fseek(arq,0,SEEK_END);
               /* Inclui novo registro */
               fwrite(&td,sizeof(td),1,arq);
          }
          printf("\nContinuar? [S/N]: ");
          scanf(" %c",&escolha);
     }while(toupper(escolha)=='S');
     fclose(arq);
     return(0);
}
```

```
/* prog33.c */
#include <stdio.h>
#include <stdlib.h>
#include "temperaturas.h"

int main(void)
{
      FILE *arq;
      struct temperatura t;

      /* Tenta abrir o arquivo para leitura */
      arq=fopen(NOMEARQUIVO,"rb");
      if(arq==NULL){
            fprintf(stderr,"Nao abriu %s\n",NOMEARQUIVO);
            exit(1);
      }
      printf("-- Data ---- Minima --- Maxima --\n");
      /* Exibe todos os registros cadastrados */
      while(fread(&t,sizeof(t),1,arq)==1 && !feof(arq))
            printf("%02d/%02d/%04d   %6.1f     %6.1f\n",
                        t.quando.dia, t.quando.mes, t.quando.ano,
                        t.minima, t.maxima);
      printf("--------------------------------\n");
      fclose(arq);
      return(0);
}
```

12. Apêndices

12.1. Modificadores de Tipo

12.1.1. Modificadores de tipo quanto ao sinal

Estes modificadores podem ser utilizados na declaração de variáveis inteiras para definir se as mesmas terão sinal ou não, afetando o intervalo de valores que elas poderão armazenar.

Modificador	Descrição
`signed`	A variável pode receber valores negativos ou positivos. *Padrão*
`unsigned`	A variável somente pode receber valores positivos.

O modificador padrão em C é o `signed`, portanto se não for definido qual é o modificador a ser utilizado, automaticamente a variável terá sinal.

Exemplo:

```
signed int ns;
unsigned int  nu;
int ni;               /* signed int */
```

12.1.2. Modificadores de tipo quanto ao tamanho

Estes modificadores podem ser utilizados com variáveis `int` para indicar o tamanho em bytes utilizado para armazenar a variável (2 ou 4).

Modificador	Descrição	Tamanho	Intervalo
`short`	Formato curto	2 bytes	-32768 a 32767
`long`	Formato longo. *Padrão*	4 bytes	-2147483648 a +2147483647

Em arquiteturas de 32 bits o modificador padrão é `long`, o que correspondente a uma palavra de máquina.

Exemplo:

```
long int nl;
short int ns;
```

12.1.3. Tabela de tipos

A partir de todas as combinação válidas de tipos de dados e seus modificadores, pode-se obter as seguintes possibilidades para a declaração de variávies numéricas, em C:

Tipo	Bytes	Intervalo	Casas Decimais
`unsigned char`	1	0 a 255	0
`signed char`	1	-128 a 127	0
`unsigned short int`	2	0 a 65535	0
`signed short int`	2	-32768 a 32767	0
`unsigned long int`	4	0 a 4294967295	0
`signed long int`	4	-2147483648 a +2147483647	0
`float`	4	3.4E-38 a 3.4E+38	7
`double`	8	1.7E-308 a 1.7E+308	15
`long double`	10	3.4E-4932 a 1.1E+4932	17

12.2. Literais

Os literais são usados para especificar valores constantes dentro de programas. A linguagem C possibilita uma grande variedade na codificação destes valores, seja quanto a base numérica utilizada ou quanto ao tamanho e formato utilizado para seu armazenamento.

Formato	Codificação	Exemplo
Decimal	Números sem 0 inicial	`10`
Octal	Prefixo `0`	`010`
Hexadecimal	Prefixo `0x`	`0xFF`
Ponto Flutuante	Dígitos decimais separados por ponto: nnn.dd	`3.14159265`
Notação Científica	*mantissa*E*expoente* (Sempre base 10)	`1.02E23`
Caracter	Um caractere entre aspas simples ou código ASCII	`'A' , 65`
String	Sequência de caracteres entre aspas duplas	`"Computação"`

12.3. Caracteres de escape

Os caracteres de escape são usados para representar alguns caracteres que, ou não estão disponíveis diretamente no teclado do computador ou em determinadas situação não podem ser inseridos diretamente dentro de um programa fonte C. Outra vantagem do uso desta forma codificada é que a portabilidade do programa é garantida. Estes caracteres podem aparecer tanto dentro de literais do tipo caracter como string.

Caracter	Descrição	Caracter	Descrição
`\b`	Retrocesso ou Backspace	`\"`	Aspa dupla
`\n`	Alimentação de linha	`\\`	Contrabarra
`\r`	Retorno do carro	`\`*nn*	Caracter cujo código ASCII, em octal, é *nn*. Ex: `'\33'` => ESC
`\t`	Tabulação horizontal (TAB)	`\0`	Caracter nulo (terminador de string)
`\'`	Aspa simples		

12.4. Operadores bit-a-bit

Estes operadores realizam operações em nível de bit com os seus operandos. Estas operações são as da álgebra booleana, a partir da representação binária dos valores processados.

Operador	Descrição
`&`	AND bit-a-bit
`|`	OR bit-a-bit
`^`	XOR bit-a-bit
`~`	NOT bit-abit

13. Exercícios

1) Escrever ao lado de cada chamada à função **printf** o que será exibido na execução deste programa. Colocar um traço (_) no lugar de espaços em branco.

```
#include <stdio.h>

int main(void)
{
      int a=54, b=2;
      char txt[]="programacao";
      float x=25.8;

      printf("[%-4d]", a);
      printf("[%02d]", b);
      printf("[%10s]", txt);
      printf("[%c]", txt[b]);
      printf("[%-5.3s]", txt);
      printf("[%8.2f]", x);
      printf("[%-5.1f]", x);
      getchar();
      return(0);
}
```

2) Escrever ao lado de cada chamada à função **printf** o que será exibido na execução deste programa. Colocar um traço (_) no lugar de espaços em branco.

```
#include <stdio.h>

int main(void)
{
      int i=1, vet[]={6, 12, 590, 31};
      double pi=3.1415;
      char curso[11]="computação";

      printf("[%x] [%d] [%o]", vet[i], vet[i], vet[i]);
      printf("[%03d]", i);
      printf("[%-6.2f]", pi);
      printf("[%7.4s]", curso);
      printf("[%c]", curso[i]);
      getchar();
      return(0);
}
```

3) Escrever ao lado de cada chamada à função **printf** o que será exibido na execução deste programa. Colocar um sublinhado (_) no lugar de espaços em branco.

```
#include <stdio.h>

int main(void)
{
      int mt[2][2] = {{32, 75}, {59,104}};
      int i = 0, j = 1;
      char str[] = "ASDFGHJKL";

      printf("[%05d]", mt[i++][j]);
      printf("[%-4d]", ++j);
      i += j;
      printf("[%04d]", i);
      str[j] = 'W';
      printf("[%6.5s]", str);
      i = (mt[0][1]>mt[1][0])? 2: 5;
      printf("[%c]", str[i]);
      getchar();
      return(0);
}
```

4) Escrever ao lado de cada chamada à função **printf** o que será exibido na execução deste programa. Colocar um sublinhado (_) no lugar de espaços em branco.

```
#include <stdio.h>

int main(void)
{
      int i=1, j=10, vet[]={2, 4, 8, 16, 32, 64};
      float vf=1.5;

      i+= (int) vf;
      printf("[%03d]", vet[++i]);
      i = sizeof(vet);
      printf("[%-4d]", i--);
      vf *= i;
      printf("[%6.2f]", vf);
      vf =  j / sizeof(j);
      printf("[%d]", j);
      printf("[%5.1f]", vf);
      getchar();
      return(0);
}
```

5) Escrever um programa que declara três variáveis **int** (a, b e c). Ler um valor maior que zero para cada variável (se o valor digitado não é válido, mostrar mensagem e ler novamente). Exibe o menor valor lido multiplicado pelo maior e o maior valor dividido pelo menor.

6) Escrever um programa que declara um vetor **float** com 10 elementos. Ler um valor para cada posição do vetor. Se o maior valor do vetor não estiver na última posição então o programa deve trocá-lo de posição com o último. Mostrar o vetor, um valor por linha.

7) Escrever um programa que declara um vetor **double** com 20 elementos. Ler um valor para cada posição do vetor. Encontrar o maior valor do vetor e mostrar todos os índices onde este valor máximo está armazenado.

8) Escrever um programa que declara e lê uma matriz **double** com 3 linhas e 4 colunas. Ler um valor para cada posição da matriz. Ler em seguida outro valor **double**. Contar e exibir quantos valores contidos na matriz são maiores e quantos são menores que o valor informado.

9) Escrever um programa que declara uma matriz **int** com 4 linhas e 6 colunas. Ler um valor para cada posição da matriz. Calcular e mostrar (com uma casa decimal) o percentual de valores pares e o percentual de valores ímpares armazenados em cada uma das linhas da matriz, exatamente no formato apresentado no exemplo abaixo.

```
A linha 00 possui 40.5% de valores pares e 59.5% de valores impares
A linha 01 possui 53.6% de valores pares e 46.4% de valores impares
```

10) Escrever uma função **int ultimodia(int mes, int ano);** que recebe um mês e um ano como parâmetro e retorna qual é o último dia daquele mês fornecido. A função deve funcionar mesmo para anos bissextos.

```
r = ultimodia(5, 1999);       /* r recebe 31 */
r = ultimodia(2, 2003);       /* r recebe 28 */
r = ultimodia(2, 2004);       /* r recebe 29 */
```

11) Escrever uma função **int fibonacci(int n);** que recebe por parâmetro um número inteiro **n** e retorna o **n**-ésimo elemento da série de Fibonacci. A série de Fibonacci é uma seqüência onde os dois primeiros valores são 1 e os demais correspondem à soma dos dois valores imediatamente anteriores (1, 1, 2, 3, 5, 8, 13, 21, ...)

```
r=fibonacci(1);               /* r recebe 1   */
r=fibonacci(2);               /* r recebe 1   */
r=fibonacci(7);               /* r recebe 13  */
```

12) Escrever uma função **int fatorial(int n);** que recebe um número inteiro positivo por parâmetro e retorna o fatorial deste número.

```
n=fatorial(8);          /* n recebe 40320  */
n=fatorial(5);          /* n recebe 120    */
```

13) Escrever um programa que lê dois valores **int** na função **main**. Somente pode ser aceito valores diferentes (enquanto forem iguais, ler novamente) Chamar uma função denominada **calcularesto**, que deve calcular e retornar o resto da divisão do maior pelo menor valor. Exibir o valor retornado, na função **main**. Não pode ser declarada nenhuma variável global.

14) Escrever um programa que lê, repetidamente um valor **int** na função **main**, até que o usuário digite 0 (zero). Para cada valor digitado, deve-se chamar uma função denominada **calculo**. Se o valor digitado for par esta função deve retornar o seu triplo. Se o valor for ímpar, a função deve retornar o seu cubo (N^3). O valor retornado deve ser exibido na função **main**. Não pode ser declarada nenhuma variável global.

```
Informe um número: 10
Resultado: 30
Informe um número: 5
Resultado: 125
Informe um número: 0
```

15) Escrever um programa que declara e lê uma variável do tipo **struct data**. Exibir o nome da estação do ano correspondente à data informada.

16) Escrever um programa que declara um vetor de **struct data** com 5 elementos, lê valores para este vetor e no final mostra qual a maior data informada.

17) Escrever ao lado de cada chamada à função **`printf`** o que será exibido na execução deste programa. Colocar um sublinhado (_) no lugar de espaços em branco.

```
#include <stdio.h>
#define FIM  12
int main(void)
{
    int n=0, k=2, z=9;
    double vals[4]={8.5, 19.35, 5.6, 7.0}, *p;
    char txt[]="MNBVCXZ";

    p = vals;
    printf("[%03d]", z++);
    vals[++n] = (double)z / 4;
    p++;
    printf("[%7.3f]", *p);
    for(n=z=0; z<FIM && n<4; n++)
        z+= (int)vals[n];
    printf("[%-4d]", z);
    k *= (n)?3:5;
    printf("[%d]", k);
    printf("[%7.3s]", txt);
    return(0);
}
```

18) Escrever ao lado de cada chamada à função **`printf`** o que será exibido na execução deste programa. Colocar um sublinhado (_) no lugar de espaços em branco.

```
#include <stdio.h>

int main(void)
{
    int valores[6]={7, 12, 19, 5, 26, 21}, a=0, i=3;
    float val = 7.95;
    int *p;
    char texto[]="compilelog";

    valores[++a] = (int) val;
    p = valores;
    *p = (i<a)?10:20;
    val+=a;
    printf("[%-7.2f]", val);
    p++;
    printf("[%-4d]", *p);
    for(i=a=0; i<4; i++)
        a+=valores[i];
    printf("[%5d]", a);
    printf("[%06d]", --i);
    printf("[%-7.5s]", texto);
    return(0);
}
```

19)Escrever ao lado de cada chamada à função **printf** o que será exibido na execução deste programa. Colocar um sublinhado (_) no lugar de espaços em branco.

```
#include <stdio.h>

int main(void)
{
    int x=17, y=4, lista[]={26, 3, 14, 19, 30};
    int *p;
    float val=0.0;
    char msg[]="LKJHGFDSA";

    y = (x)?2:1;
    printf("[%04d]", lista[++y]);
    p = lista;
    x += *p;
    printf("[%5d]", x--);
    p++;
    printf("[%-4d]", *p);
    *p = 21;
    val =  lista[1] / (y - 1);
    printf("[%-8.4f]", val);
    msg[1] = 'X';
    printf("[%-8.4s]", msg);
    return(0);
}
```

20)Completar o programa abaixo, sem usar variáveis globais, escrevendo as seguintes funções:

- l**er_num**: esta função deve ler e retornar um valor **float** maior que zero. Se o valor digitado não for válido, mostrar mensagem e ler novamente.
- **calculo**: esta função recebe a quantidade vendida e o preço unitário por parâmetro e retorna o preço total de venda (quantidade multiplicado pelo preço unitário, menos desconto de 10%) e armazena no terceiro parâmetro, passado por referência, o desconto concedido sobre a compra (10% do total bruto).

```
#include <stdio.h>
int main(void)
{
   float quant, unitario, final, desc;

   printf("Quantidade vendida: ");
   quant = ler_num();
   printf("Preco unitario: ");
   unitario = ler_num();
   final = calculo(quant, unitario, &desc);
   printf("Valor a pagar: R$ %.2f\n", final);
   printf("Desconto concedido: R$ %.2f\n", desc);
   return(0);
}
```

21)Completar o programa abaixo, escrevendo a função **`int tempo(int n, int *hora, int *min);`** Ela recebe, por valor, um parâmetro **n** indicando a quantidade de minutos transcorridos após a meia-noite. Se este valor for menor que zero ou maior do que a quantidade de minutos existente em um dia, a função deve retornar 0 (zero). Senão, a função deve calcular qual a hora e minuto correspondente àquele horário e armazenar, nos parâmetros **hora** e **min**, passados por referência e retornar 1 (um). Exemplo de resultados: 100 minutos após a meia-noite corresponde a 01:40 h; 35 minutos após a meia-noite corresponde são 00:35 h; 151 minutos após a meia noite são 02:31 h. Não pode ser declarada nenhuma variável global.

```
#include <stdio.h>

int tempo(int n, int *hora, int *min);

int main(void)
{
      int q, h, m, r;
      printf("Quantos minutos depois da meia-noite? ");
      scanf("%d", &q);
      r=tempo(q, &h, &m);
      if(r!=0)
            printf("%02d:%02d", h, m);
      else
            printf("Inválido!");
      fflush(stdin); getchar();
      return(0);
}
```

22)Completar o programa abaixo, escrevendo a função **int convertebytes(int b, float *kb, float *mb, float *gb);** Ela recebe por parâmetro (por valor) um número inteiro **b**, indicando uma quantidade de bytes. Se este valor for menor que zero, a função deve retornar 0 (zero). Se o valor maior que zero, a função deve converter este valor para kilobytes, megabytes e gigabytes e armazenar estes valores, respecivamente, nos parâmetros **kb**, **mb** e **gb**, passados por referência e retornar 1 (um). Não pode ser declarada nenhuma variável global.

```
#include <stdio.h>
int convertebytes(int b, float *kb, float *mb, float *gb);
int main(void)
{
      int b, teste;
      float k, m, g;

      printf("Quantidade em bytes? ");
      scanf("%d", &b);
      teste=convertebytes(b, &k, &m, &g);
      if(teste==0)
            printf("Valor inválido!");
      else
            printf("%d bytes = %.2f Kb,  %.2f Mb, %.2f Gb\n", b, k, m, g);

      fflush(stdin); getchar();
      return(0);
}
```

23)Escrever um programa que declara um vetor **`int`** com 20 elementos na função **`main`** e lê um valor para cada posição do vetor na mesma função. Em seguida exibir o conteúdo do vetor em uma função chamada **`exibe`**, seguindo a seguinte lógica: Se a soma dos valores armazenados no vetor for um número par, mostrar o vetor a partir do primeiro elemento até o último. Se a soma dos valores armazenados no vetor for um número impar, mostrar o vetor a partir do último elemento até o primeiro. O programa não pode utilizar nenhuma variável global.

24)Escrever um programa que declara um vetor **`int`** com 40 elementos na função **`main`** e lê um valor para cada posição do vetor na mesma função. A seguir chama uma função **`containpares`** passando o vetor e seu tamanho por parâmetro. A função deve calcular e retornar a quantidade de valores impares armazenados no vetor. O valor retornado deve ser exibido na função **`main`**. O programa não pode utilizar nenhuma variável global.

25) Escrever uma função **`int posdif(char *str, char c);`** que recebe como parâmetro a string **`str`**, passada por referência e o caracter **`c`**, passado por valor. A função deve calcular e retornar a soma das posições de todos os caracteres em **`str`** que são diferentes de **`c`**.

```
n = posdif("aula", 'a');   /* n recebe 3 (1+2) */
n = posdif("aula", 'u');   /* n recebe 5 (0+2+3) */
n = posdif("aula", 'x');   /* n recebe 6 (0+1+2+3) */
```

26) Escrever uma função **`int multmm(int *vetor, int tamanho);`** que recebe como parâmetros um vetor **`int`** e o seu tamanho. A função deve retornar o maior valor do vetor multiplicado pelo menor.

```
int v[5]={4, 2, 5, 10, 6}, n;
n=multmm(v, 5);  /* n=20 (10*2) */
```

27) Escrever um programa que inicialmente lê um número inteiro **`N`**. Se o valor informado for menor que 2 (dois) finalizar o programa. Caso contrário, alocar dinamicamente um vetor **`int`** com o tamanho informado. Ler um valor para cada posição do vetor e, no final, mostrar se o vetor possui algum valor repetido, isto é, que foi informado mais de uma vez. Atenção: os valores não precisam se exibidos, apenas a informação **`"Existe valor repetido no vetor"`** ou **`"Não existe valor repetido no vetor"`**.

28) Escrever um programa que inicialmente lê um número inteiro **`N`**, que deve ser maior que 1 (um). Se o número informado não atender este requisito, deve ler novamente até ser informado um valor válido. Em seguida, o programa deve alocar dinamicamente um vetor **`int`** com o tamanho informado. Ler um valor para cada posição do vetor. No final, mostrar quantos valores do vetor são divisíveis por **N**.

29) Escrever uma função **`int diadoano(struct data *dt);`** que recebe uma data por parâmetro e retorna o número de dias transcorridos desde o início do ano até a data fornecida (contando inclusive a data fornecida). Assumir que a data fornecida como parâmetro é válida.

```
struct data d1={1, 1, 2000}, d2={15, 3, 2003}, d3={31, 12, 2004};
int res;

res = diadoano(&d1);    /* res recebe 1   */
res = diadoano(&d2);    /* res recebe 74  */
res = diadoano(&d3);    /* res recebe 366 */
```

30) Escrever uma função **`int datecmp(struct data *dt1, struct data *dt2);`** que recebe duas datas por parâmetro e retorna 0 (zero) se as datas são iguais, um valor menor que zero se a primeira data (dt1) é anterior à segunda (dt2) e um valor maior que zero se a primeira data é posterior à segunda.

31) Escrever ao lado de cada chamada à função **printf** o que será exibido na execução deste programa. Substituir espaços em branco por um traço.

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

int main(void)
{
    int n=1, *p, vet[]={29, 16, 8, 20};
    char a[50]="boas-", b[50]="ferias!";
    float val=2.6;

    p=vet;
    vet[1] += (int) val;
    p++;
    printf("[%c]", b[n]);
    printf("[%04d]", *p);
    n=(isalpha(b[1]))?strlen(a): strlen(b);
    printf("[%3d]", --n);
    strcat(a, b);
    printf("[%9.7s]", a);
    val+=sizeof(char);
    printf("[%-5.1f]", val);
    return(0);
}
```

32)Escrever ao lado de cada chamada à função **printf** o que será exibido na execução deste programa. Substituir espaços em branco por um traço.

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

struct carro{
       char modelo[50];
       float potencia;
};

int main(void)
{
    struct carro a4={"Audi A4", 170.0};
    char s1[50]="--", s2[50]="V6 2.4";
    int i, j=2;
    float v;

    strcpy(s1, a4.modelo);
    for(i=0; s1[i]!='\0'; i++){
        if(!isalpha(s1[i]))
            j*=2;
    }
    printf("[%04d]", j);
    strcat(s1, s2);
    i = strlen(s1);
    printf("[%-3d]", i);
    printf("[%5.2s]", s2);
    printf("[%7.1f]", a4.potencia);
    v = (isupper(s1[1]))?20.5 :98.7;
    printf("[%6.2f]", v);
    return(0);
}
```

33)Escrever uma função **`int ltrim(char *s1, char *s2);`** que recebe duas string por parâmetro e copia o conteúdo da string **`s2`**, sem os espaços em brancos do início (esquerda) para a string **`s1`**. Retornar o número de caracteres que foram copiados.

```
char a[100], b[100]="    Prova de C  ";
int n;
n=ltrim(a, b);
printf("A=[%s] N=%d", a, n);  /* A=[Prova de C  ] N=12 */
```

34)Escrever um programa que declara uma string com 80 caracteres. A seguir o programa deve ler repetidamente esta string e mostrar o seu tamanho (número de caracteres ocupados) até que o valor digitado seja "**`FIM`**". No final mostrar quantas leituras foram feitas e o número total de caracteres.

```
Digite uma string: AULA
Caracteres: 4
Digite uma string: TESTANDO
Caracteres: 8
Digite uma string: FIM
Caracteres: 3
Strings Lidas: 3    Total de Caracteres: 15
```

35)Escrever uma função **`void repeatstr(char *s1, char *s2, int num);`** que copia o conteúdo da string **`s2`** para a string **`s1`**, repetindo-o tantas vezes quantas definidas no parâmetro **`num`**. Por exemplo:

```
char txt[100];
repeatstr(txt, "AULA", 2);  /* txt recebe "AULAAULA"   */
repeatstr(txt, "*-", 3);    /* txt recebe "*-*-*-"     */
repeatstr(txt, "-", 10);    /* txt recebe "----------" */
```

36) Escrever a função **`int lastpos(char *str, char c);`** que recebe uma strings (str) e um caracter (c) por parâmetro. A função deve retornar o último índice na string em que se encontra o caracter fornecido. Se o caracter não estiver na string, retornar -1.

```
r=lastpos("aula", 'a');       /* retorna 3  */
r=lastpos("exemplo", 'x');    /* retorna 1  */
r=lastpos("exemplo", '@');    /* retorna -1 */
```

37) Escrever um programa (tabuada.c) que recebe um número como argumento de linha de comando e exibe a tabuada correspondente ao número fornecido (produtos do número por 1 até 10), um produto por linha.

```
$ ./tabuada 5

 1 x 5 =  5
 2 x 5 = 10
    ...
 9 x 5 = 45
10 x 5 = 50
```

38) Escrever um programa (letras.c) que recebe o nome de um arquivo tipo texto como argumento de linha de comando. Exibir o nome do arquivo, o percentual de letras e o percentual de outros caracteres armazenados no arquivo. Exibir os valores com 1 casa decimal. Qualquer mensagem de advertência ao usuário (número de argumentos incorretos, não conseguiu abrir arquivo, etc.) deve ser exibida na saída de erros. Seguir exatamente o modelo de saída abaixo:

```
$ ./letras exemplo.txt
exemplo.txt possui 73.2% de letras e 26.8% de outros caracteres
```

39) Escrever um programa (separa.c) que recebe o nome de três arquivos como argumento na linha de comando. O programa deve ler o conteúdo do primeiro arquivo, gravando as letras e espaços no segundo arquivo e os demais caracteres no terceiro arquivo. No final mostrar o número de caracteres que foram escritos em cada arquivo, conforme o formato abaixo. Qualquer mensagem de advertência ao usuário (número de argumentos incorretos, não conseguiu abrir arquivo, etc.) deve ser exibida na saída de erros.

```
$ ./separa aula.txt letnum.txt outros.txt
Copiou 348 caracteres para letnum.txt e 59 caracteres para outros.txt
```

40) Escrever um programa que recebe um nome de arquivo como argumento de linha de comando, abre o arquivo, conta e exibe quantas letras maiúsculas e quantas letras minúsculas o arquivo contêm. Qualquer mensagem de advertência ao usuário (número inválido de argumentos, não conseguiu abrir o arquivo,...) deve ser exibida na saída de erros.

```
$ ./conta aula.txt
conta.txt possui 18 letras maíusculas e 25 letras minúsculas.
```

41) Escrever um programa (copiamin.c) que recebe o nome de dois arquivos tipo texto pela linha de comando e copia o conteúdo do segundo arquivo para o primeiro convertendo todas as letras para minúsculo. Mostrar, no final, o número total de caracteres que foram copiados. Qualquer mensagem de advertência ao usuário (número de argumentos incorretos, não conseguiu abrir arquivo, etc.) deve ser exibida na saída de erros.

```
$ ./copiamin teste1.txt teste2.txt
Foram copiados 569 caracteres de arq1.txt para arq2.txt
```