UFU – Universidade Federal de Uberlândia

Faculdade de Computação

Apostila de Linguagem C
(Conceitos Básicos)

Prof. Luiz Gustavo Almeida Martins

LINGUAGEM C – Conceitos Básicos

1. INTRODUÇÃO

1.1. HISTÓRICO

A Linguagem C, criada em 1970 por Dennis Ritchie, é uma evolução da Linguagem B que, por sua vez, foi uma adaptação feita, a partir da Linguagem BCPL, por Ken Thompson. Ela é estreitamente associada ao sistema operacional UNIX, já que as versões atuais do próprio sistema foram desenvolvidas utilizando-se esta linguagem.

Devido ao crescente uso e interesse da comunidade de computação pela linguagem, em 1983, o American National Standards Institute (ANSI), estabeleceu um comitê para prover uma definição moderna e abrangente da linguagem C. O resultado desta comissão foi uma padronização denominada ANSI-C em 1988. A maior contribuição deste trabalho foi a incorporação de uma biblioteca padrão, presentes em todas as variações/versões da linguagem, que fornece funções de acesso ao sistema operacional, entrada e saída formatada, alocação de memória, manipulação de strings (cadeias de caracteres), etc.

1.2. CONCEITOS BÁSICOS

A filosofia básica da linguagem C é que os programadores devem estar cientes do que estão fazendo, ou seja, supõe-se que eles saibam o que estão mandando o computador fazer, e explicitem completamente as suas instruções. Assim, ela **alia a elegância e a flexibilidade das linguagens de alto nível** (ex: suporte ao conceito de tipo de dados) **com o poderio das linguagens de baixo nível** (ex: manipulação de bits, bytes e endereços).

O C é uma linguagem de programação de finalidade geral, utilizada no desenvolvimento de diversos tipos de aplicação, como processadores de texto, sistemas operacionais, sistemas de comunicação, programas para solução de problemas de engenharia, física, química e outras ciências, etc.

O código-fonte de um programa C pode ser escrito em qualquer **editor de texto que seja capaz de gerar arquivos em código ASCII** (sem formatação). Como o ambiente de programação utilizado (Turbo C) é para o sistema operacional DOS, estes arquivos devem ter um **nome de no máximo 8 caracteres** e **a extensão "c"** (exemplo: NONAME.C).

Após a implementação, **o programa-fonte** (um ou mais arquivos-fonte) é **submetido aos processos de compilação e linkedição** para gerar o programa executável (com extensão "exe"). Durante o processo de compilação, **cada arquivo-fonte é compilado separadamente**, produzindo um arquivo de código-objeto com a extensão "obj". Estes arquivos-objeto contêm instruções em linguagem de máquina (códigos binários) entendidas somente pelos microprocessadores. Na linkedição, **todos os arquivos-objetos pertencentes ao projeto, bem como as bibliotecas declaradas nos códigos-fonte são processadas em conjunto**, visando a produção do arquivo executável correspondente.

Normalmente, tanto o arquivo-objeto quanto o arquivo executável possuem o mesmo nome do arquivo-fonte. Entretanto, quando desejado, o usuário poderá definir diferentes nomes para cada tipo de arquivo.

1.3. CARACTERÍSTICAS GERAIS

A linguagem C possui as seguintes características:

- **Alta portabilidade** inerente da padronização ANSI, ou seja, é possível tomar um código-fonte escrito para uma máquina, compilá-lo e rodá-lo em outra com pouca ou nenhuma alteração;

- Gera **programas formados basicamente por funções**, o que **facilita a modularização e a passagem de parâmetros** entre os módulos;

- Inicia a execução a partir da **função main()**, necessária em todos os programas;

- Uso de **chaves** (**{ }**) para agrupar comandos pertencentes a uma estrutura lógica (ex: if-else, do-while, for, etc.) ou a uma função;

- Uso do **ponto e vírgula** (**;**) ao final de cada comando;

- É "**case sensitive**", ou seja, **o compilador difere maiúsculas de minúsculas**. Assim, se declarar uma variável de nome *idade*, esta será diferente de *Idade*, *IDADE*, etc. Além disso, **todos os comandos da linguagem devem ser escritos em minúsculo**.

1.4. ESTRUTURA BÁSICA DE UM PROGRAMA C

```
Diretivas
Declarações de variáveis/constantes globais;
Prototipação de funções;
tipo_dado main(lista_parametros) /* Função Principal e Obrigatória */
{
        Declarações de variáveis/constantes locais;
        Comandos;
        Estruturas de controle (seleção e/ou repetição);
        Comentários;
        Chamada a outras funções;
}
tipo_dado func1(lista_parametros)
{
        Bloco de comandos;
}
tipo_dado func2(lista_parametros)
{
        Bloco de comandos;
}
•
•
•
tipo_dado funcN(lista_parametros)
{
        Bloco de comandos;
}
```

1.5. COMENTÁRIOS

Em C, comentários podem ser escritos em qualquer lugar do texto para facilitar a interpretação do algoritmo. Para que o comentário seja identificado como tal, **ele deve ter um /* antes e um */ depois**.

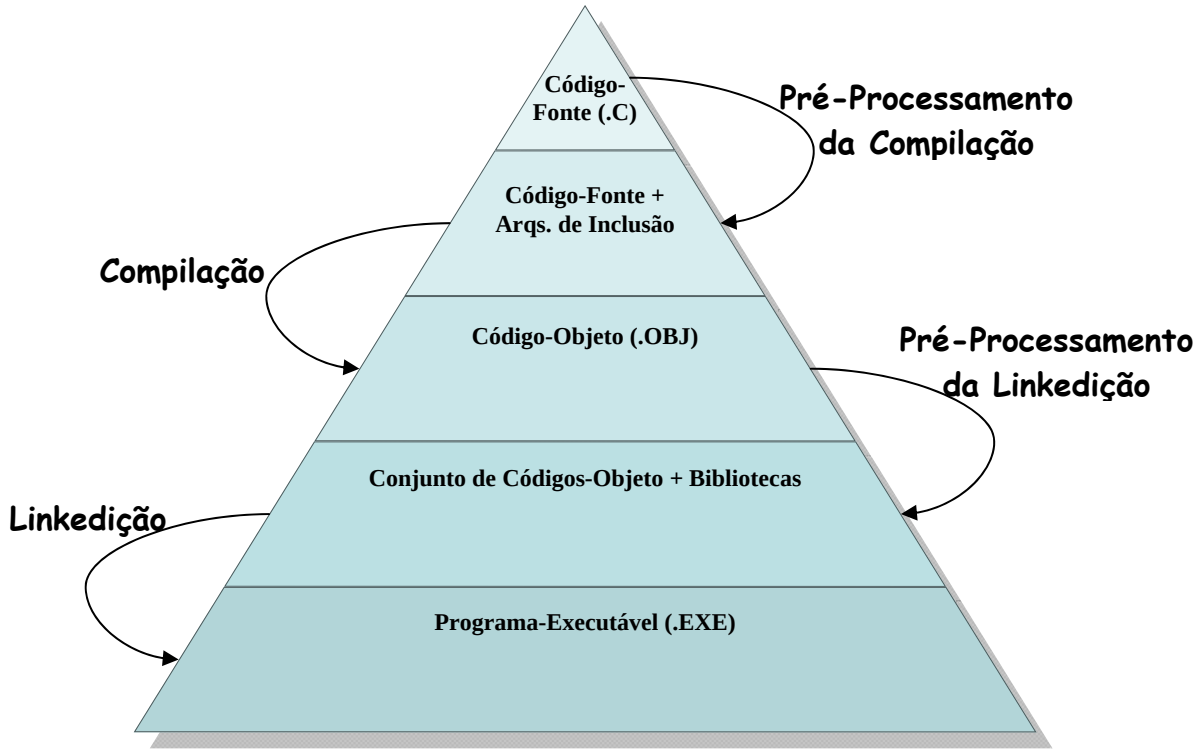
Vale lembrar que, durante a compilação do programa, os comentários são descartados pelo compilador.

Exemplo:

/* esta e´ uma linha de comentário em C */

2. DIRETIVAS

Durante a confecção do código-fonte, além dos comandos normais, o programador poderá utilizar-se de diretivas de compilação. As diretivas são prontamente reconhecidas, vista que aparecem quase sempre no início dos arquivos-fonte, **com o prefixo # seguido pelo nome da diretiva e seus argumentos**. Ao contrário dos comandos normais, **não se usa ponto e vírgula ao final da declaração da diretiva**. Elas direcionam o compilador entre vários caminhos durante a fase de pré-processamento.



ETAPAS DO PROCESSO DE COMPILAÇÃO/LINKEDIÇÃO

Um importante exemplo é a diretiva **INCLUDE**, a qual permite **incorporar, no arquivo-fonte, o conteúdo de outro arquivo** passado como argumento. Ela propicia a utilização de arquivos de inclusão criados pelo usuário ou fornecidos junto com o compilador (normalmente arquivos de cabeçalho com extensão “h”). Os arquivos criados pelos usuários visam a reutilização de funções de uso geral já desenvolvidas, de modo a evitar digitações repetitivas. Já os arquivos fornecidos pelo fabricante do compilador visam suprir as definições e declarações mais frequentemente necessárias, de acordo com a norma ANSI-C, possibilitando, assim, a portabilidade do código-fonte. As principais formas de indicar a localização do arquivo de inclusão são:

#include <stdio.h> Busca o arquivo stdio.h primeiramente no diretório de inclusão (padrão: C:\TC\INCLUDE)

#include "stdio.h" Busca o arquivo stdio.h primeiramente no diretório de trabalho.

Independentemente da forma utilizada, este comando ordena ao pré-processador que carregue o conteúdo do arquivo *stdio.h* como parte integrante do arquivo-fonte, exatamente naquele ponto. Tal ação não afeta fisicamente o arquivo-fonte.

Os arquivos de inclusão podem ser agrupados, isto é, um arquivo de inclusão pode conter outros arquivos de inclusão adicionais, e assim por diante, até o limite de pelo menos 8 níveis, conforme determinação do padrão ANSI (no Turbo C pode haver até 16 níveis de inclusões aninhadas).

Na tabela abaixo são apresentados alguns arquivos de inclusão que podem ser utilizados na confecção dos programas.

ARQUIVOS DE CABEÇALHO – PADRÃO ANSI C

ARQUIVO	FINALIDADE
assert.h	Contém a macro "assert" que é usada para incluir diagnóstico em programas
ctype.h	Declara funções para testar caracteres
errno.h	Define o número de erro do sistema
float.h	Define os valores em ponto flutuante específicos do compilador
limits.h	Define tipos de valores-limites específicos do compilador
locale.h	Utilizada para adaptar as diferentes convenções culturais
math.h	Define as funções matemáticas e macros usadas para operações matemáticas em linguagem C
setjmp.h	Provê uma forma de evitar a seqüência normal de chamada e retorno de função profundamente aninhada
signal.h	Provê facilidades para manipular condições excepcionais que surgem durante a execução, como um sinal de interrupção de uma fonte externa ou um uso na execução
stdarg.h	Define macros para acessar argumentos quando uma função usa um número variável de argumentos
stddef.h	Define funções, constantes, macros e estruturas usadas para operações padrões de entrada e saída
stdio.h	Contém funções, tipos e macros de entrada e saída
stdlib.h	Contém funções para conversão de números, alocação de memória e tarefas similares
string.h	Define funções para manipulação de *"strings"*
time.h	Define funções e estruturas usadas na manipulação de data e hora.

Outra diretiva que também muito utilizada é a **DEFINE**. Ela é o mecanismo básico do C para **criar macros e identificadores**. A diretiva DEFINE é precedida do símbolo # e seguida de dois argumentos, de acordo com a finalidade desejada.

Na criação de um identificador, o primeiro argumento indica o **nome do identificador** e o segundo define o **conteúdo** que será associado a este identificador, como no exemplo abaixo:

#define NOME "Fulano de Tal"

Neste caso, o pré-processador da compilação substituirá toda a aparição subseqüente do identificador NOME, em qualquer lugar do código-fonte, pela string "Fulano de Tal". Vale ressaltar que, se o pré-processador encontrar a palavra NOME dentro de uma string, isto é, entre aspas (ex: *"Digite o seu NOME"*), não será realizada nenhuma substituição.

Quando a string é muito longa para ocupar uma única linha, pode-se empregar o caracter especial de escape (barra invertida - \) antes do <ENTER> e continuar digitando o texto restante na próxima linha, como a seguir:

**# define AVISO "Este texto é muito grande, **
assim estou utilizando a barra invertida para mantê-lo dentro da minha tela"

Não existe nenhuma regra de substituição de strings. Assim, qualquer seqüência de caracteres será literalmente e exatamente inserida no código-fonte, onde quer que o identificador correspondente seja encontrado. A única exceção é o macete de continuação de linha, já que a barra invertida não é parte integrante da string. A verificação sintática (sentido contextual) da substituição será realizada posteriormente durante a fase de compilação propriamente dita.

Na criação de macros, a diretiva DEFINE funciona de modo similar a uma função, exceto pelo fato de que, como macro, ela permite que argumentos sejam suprimidos, ao contrário das funções. O primeiro argumento da diretiva indica **o nome da macro e os argumentos que serão utilizados por ela**. Já o segundo argumento define **como os argumentos da macro serão aplicados** quando o pré-processador encontrar o nome da macro no código-fonte restante. Abaixo é apresentado um exemplo de macro:

#define CUBO(X) ((X)*(X)*(X))

Neste exemplo, o pré-processador substituirá o texto CUBO(X) pelo valor correspondente a X^3, como nos exemplos abaixo:

CUBO(2) ((2)*(2)*(2)) = 8

CUBO(A+B) ((A+B)*(A+B)*(A+B))

Neste último caso, é possível notar a importância dos parênteses em torno X na definição da ação na declaração do DEFINE. Sem eles, a substituição de CUBO(A+B) seria equivocada, visto que o resultado seria (A+B*A+B*A+B) e a multiplicação tem precedência sobre a soma.

O ANSI-C fornece dois operadores do pré-processador para serem utilizados em macros definidas pelo DEFINE. São eles:

\# transforma o argumento que ele precede em uma string entre aspas

Ex: **#define mkstr(s) # s**

printf(mkstr(Processamento de Dados));

concatena duas palavras

Ex: **#define concat(a,b) a ## b**

printf(concat("Processamento"," de Dados");

Em ambos os casos, o resultado gerado é equivalente ao comando:

printf("Processamento de Dados");

Apesar de existirem outras diretivas de compilação, inclusive algumas que permitem compilação condicional, essas fogem ao escopo do curso e, portanto, não serão discutidas neste material.

3. PROTÓTIPOS DE FUNÇÕES

O padrão ANSI C expandiu a declaração tradicional de função, permitindo que **a quantidade e os tipos dos argumentos das funções sejam declarados no início do programa**. Esta definição expandida é chamada de ***protótipo da função***.

Protótipos permitem que o C **forneça uma verificação mais forte de tipos**, ou seja, através dos protótipos, o C pode encontrar e apresentar quaisquer conversões ilegais de tipo entre o argumento usado para chamar a função (argumento atual) e aquele utilizado na definição de seus parâmetros (argumento formal). Além disso, o uso de protótipo também permite **encontrar diferenças entre o número destes argumentos**. Portanto, a sua utilização ajuda na detecção de erros antes que eles ocorram.

A forma geral de uma definição de protótipo de função é:

tipo nm_função (tipo nm_arg1, tipo nm_arg2, ... , tipo nm_argN);

Exemplo:

```
# include <stdio.h>
unsigned int NUM;
int fat (unsigned int NRO); /* Protótipo da Função fat() */
void main (void)
{
        scanf("%u", &NUM);
        printf("\nO fatorial de %u é %d", NUM, fat(NUM));
}
Int fat(unsigned int NRO)
{
        if NRO < 2
                return (1);
        else
                return (NRO * fat(NRO-1));
}
```

O uso dos nomes dos parâmetros é opcional. Porém, eles habilitam o compilador a identificar qualquer incompatibilidade de tipo através do nome quando ocorre um erro.

Devido à necessidade de compatibilidade com a versão original do C, a qual não possui protótipo de funções, algumas informações adicionais devem ser consideradas:

- Quando uma função não possui argumentos, seu protótipo deve usar **void** dentro dos parênteses (ex: *int func (**void**);*). Isso infroma ao compilador que a função não tem argumentos e qualquer chamada à função com parâmetros é um erro.

- Quando uma função sem protótipo é chamada, todos os **char** são convertidos em **int** e todos os **float** são convertidos em **double**. **Tais conversões**, que estão relacionadas com as características do ambiente original do C, **acarretam uma alocação maior de memória**. No entanto, se a função tem protótipo, **os tipos especificados na declaração do protótipo são mantidos**.

Embora não seja um erro criar programas sem protótipo de funções, o padrão ANSI recomenda fortemente o seu uso.

4. TIPOLOGIA DE DADOS

Durante a compilação do código-fonte, **o compilador C procura alocar o espaço de memória necessário para a execução do programa**. Embora o compilador possa determinar os tipos de dados e espaço de memória exigidos para as constantes encontradas no código, no caso das variáveis, isto só é possível se houver **a declaração prévia do tipo de dado empregado**.

Cada tipo de dado tem uma exigência pré-determinada do tamanho de memória e uma faixa associada de valores permitidos. Por exemplo, um caracter ocupa geralmente 1 byte, enquanto que um inteiro tem normalmente 2 bytes. Entretanto, para garantir a portabilidade do código C, esta suposição não pode ser tomada como verdadeira. Isto porque, **o tamanho e a faixa desses tipos de dados variam de acordo com o tipo de processador e com a implementação do compilador C**. O padrão ANSI estipula apenas a faixa mínima de cada tipo de dado e não o seu tamanho em bytes.

4.1. TIPOS BÁSICOS

Há 5 tipos básicos em C, conforme tabela abaixo. Todos os demais tipos são variações destes 5 tipos.

TIPOS BÁSICOS DE DADOS DO C

Tipos	Descrição	Tamanho Aprox.	Faixa Mínima
char	Caractere	8 bits = 1 byte	-127 a 127
int	Inteiro	16 bits = 2 bytes	-32.767 a 32.767
float	Ponto flutuante	32 bits = 4 bytes	7 dígitos de precisão
double	Ponto flutuante de precisão dupla	64 bits = 8 bytes	10 dígitos de precisão
void	Sem valor		

Elementos do tipo **char** são normalmente utilizados para conter valores definidos pelo conjunto de caracteres ASCII. Elementos do tipo **int** geralmente correspondem ao tamanho natural de uma palavra do computador hospedeiro. Assim, numa máquina de 16 bits, int provavelmente terá 16 bits. Numa máquina de 32, int deverá ter 32 bits. O formato exato dos elementos em ponto flutuante depende de como eles são implementados. A faixa dos tipos **float** e **double** é dada em dígitos de precisão. Já suas grandezas dependem do método usado para representar os números em ponto flutuante. Qualquer que seja o método, o número é muito grande (o padrão ANSI especifica a faixa mínima de $1x10^{-37}$ a $1x10^{+37}$). O tipo **void** declara explicitamente uma função que não retorna valor algum ou cria ponteiros genéricos.

4.2. MODIFICADORES DE TIPODE DADOS

Com exceção dos tipos **void** e **float**, os demais tipos básicos podem ser adaptados mais precisamente às necessidades do problema estudado. Para isto, é necessário preceder o tipo básico do dado com um dos modificadores de tipo (***signed***, ***unsigned***, ***long*** e ***short***), visando alterar o significado do tipo.

Normalmente, ao tipo **double** pode-se aplicar apenas o modificador ***long***. Esta modificação gera um tipo de ponto flutuante com maior precisão. Os quatro modificadores podem ser aplicados a inteiros. Os modificadores ***short*** e ***long*** visam prover tamanhos diferentes de inteiros (inteiros menores – **short int** ou maiores – **long int**). Na verdade, cada compilador é livre para escolher tamanhos adequados para o seu próprio hardware, com a única restrição de que **short int** e **int** devem ocupar pelo menos 16 bits e **long int** pelo menos 32 bits. Além disso, **short int** não pode ser maior que **int**, que, por sua vez, não pode ser maior que **long int**. O modificador **unsigned** serve para especificar variáveis sem sinal. Um **unsigned int** será um inteiro que assumirá apenas valores positivos. Já o uso de ***signed*** para o tipo **int**, apesar de permitido, é redundante. A seguir

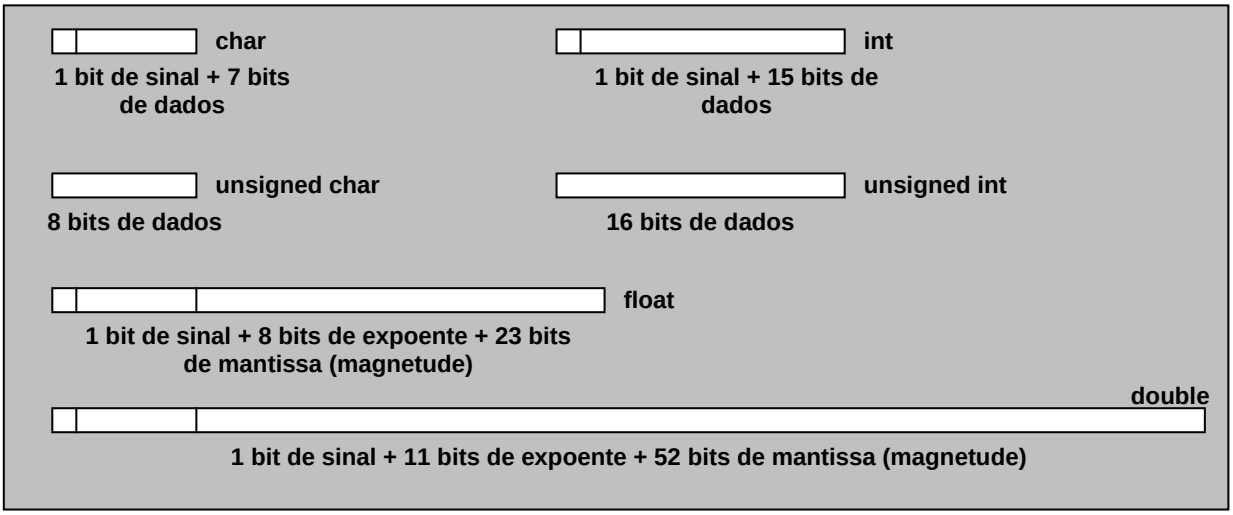
estão listados os tipos de dados permitidos e seu valores máximos e mínimos em um compilador típico para um hardware de 16 bits. Também nesta tabela está especificado o formato que deve ser utilizado para ler os tipos de dados com a função ***scanf()***:

Tipos de Dados Modificados

Tipo	Num de bits	Formato para leitura com scanf	Intervalo	
			Inicio	Fim
Char	8	%c	-128	127
unsigned char	8	%c	0	255
Signed char	8	%c	-128	127
Int	16	%i ou %d	-32.768	32.767
unsigned int	16	%u	0	65.535
signed int	16	%i ou %d	-32.768	32.767
Short int	16	%hi ou %hd	-32.768	32.767
unsigned short int	16	%hu	0	65.535
signed short int	16	%hi ou %hd	-32.768	32.767
Long int	32	%li ou %ld	-2.147.483.648	2.147.483.647
signed long int	32	%li ou %ld	-2.147.483.648	2.147.483.647
unsigned long int	32	%lu	0	4.294.967.295
Float*	32	%f, %e ou %g	3,4E-38	3.4E+38
Double*	64	%lf, %le ou %lg	1,7E-308	1,7E+308
long double*	80	%Lf	3,4E-4932	3,4E+4932

** É importante observar que os intervalos de ponto flutuante, na tabela acima, estão indicados em faixa de expoente, mas os números podem assumir valores tanto positivos quanto negativos.*

Por fim, dependendo da implementação de compilador utilizada, **poderá existir outros tipos de dados** (expandidos ou adicionais), como por exemplo, o uso do **unsigned** para os **tipos de ponto flutuante**. Porém, o uso de tipos de dado não definidos pelo padrão ANSI C **reduz a portabilidade de seu código** e, portanto, não é recomendado.



Representação Básica de Armazenamento por Tipo de Dados

4.3. MODIFICADORES DE TIPO DE ACESSO

O padrão ANSI introduziu dois modificadores de tipo que **controlam a maneira como as variáveis podem ser acessadas ou modificadas**. Esses modificadores são ***const*** e ***volatile***. Estes modificadores devem preceder os modificadores de tipo de dado e/ou tipo de dados e o nome das variáveis que eles modificam.

Variáveis do tipo *const* não podem ser modificadas pelo programa, podendo, entretanto, receber um valor inicial. Por exemplo:

const int pi = 3.14;

O modificador ***volatile*** é usado para informar ao compilador que **o valor de uma variável pode ser alterado de maneira não explicitamente especificada pelo programa**. Por exemplo, um endereço de uma variável global pode ser passado pra a rotina de relógio do sistema operacional e usado para guardar o tempo real do sistema. Nessa situação, o conteúdo da variável é alterado sem nenhum comando de atribuição explícito no programa.

É possível usar ***const*** e ***volatile*** juntos. Por exemplo, se 0x30 é assumido como sendo o valor de uma porta que é mudado apenas por condições externas, a declaração seguinte é ideal para prevenir qualquer possibilidade de efeitos colaterais acidentais.

const volatile unsigned char *porta = 0x30;

Vale ressaltar que, qualquer valor que comece com **0x** corresponde a um **constante hexadecimal**, bem como valores iniciados com **0** equivalem a **constantes octais**. Assim, para o compilador C, o valor 015 é diferente de 15.

4.4. IDENTIFICADORES NO C

O padrão ANSI define nomes de variáveis, funções, rótulos e vários outros objetos definidos pelo usuário como ***identificadores***. Esses identificadores podem variar de um a vários caracteres, sendo que **o primeiro deve ser uma letra ou um caracter especial "_" e os demais caracteres devem ser letras, números ou o caracter especial "_"**.

Vale ressaltar que, assim como na pseudolinguagem, o identificador **NÃO PODE TER espaço** nem nenhum outro caracter especial, inclusive letras acentuadas.

Os identificadores podem ter qualquer tamanho. Entretanto, de acordo com o uso do identificador (interno ou externo – chamado por outros programas), **ele deve se diferenciar dos demais** (possuir um nome diferente de todos os outros identificadores utilizados) **dentro de uma margem pré-estabelecida (6 caracteres para nomes externos e 31 caracteres para nomes internos)**.

Além disso, um identificador não pode ser igual a uma palavra-chave de C ou a um nome de função já criada pelo usuário ou existente na bliblioteca padrão.

OBS: Lembre-se que o C é case-sensitive. Portanto, letras minúsculas e maiúsculas são tratadas diferentemente (ex: cont, CONT e Cont são três identificadores diferentes).

4.5. DECLARAÇÃO DE VARIÁVEIS

Como já visto, uma variável é uma posição de memória de um determinado tipo de dado e que possui um nome, que é usada para guardar valores que podem sofrer mudanças durante a execução do programa.

Todas as variáveis do C devem ser declaradas antes de serem usadas. A forma geral de uma declaração é:

tipo lista_de_variáveis;

Sendo:

Tipo: um tipo de dado válido em C mais quaisquer modificadores; e

Lista_de_variáveis: um ou mais nomes de identificadores separados por vírgula.

Exemplos:

int nro, idade, índice;

short int idade;

double salario, pagamento;

As variáveis podem ser declaradas em três lugares básicos: dentro de funções, na definição dos parâmetros das funções e fora de todas as funções. Estas são, respectivamente, variáveis locais, parâmetros formais e variáveis globais.

VARIÁVEIS LOCAIS

As variáveis locais, também conhecidas com variáveis automáticas (pois pode utilizar o especificador ***auto*** para declará-las), **são aquelas declaradas dentro das funções** e, portanto, **só são reconhecidas pela função onde foi declarada** (lembre-se que o corpo de uma função é delimitado por chaves).

Variáveis locais existem apenas enquanto o bloco de comando em que foram declaradas está sendo executado. Isto é, uma variável local é criada na entrada de seu bloco e destruída na saída.

Ex:
```
void func1()
{
        int NRO;
        NRO = 100;
}
void func2()
{
        int NRO;
        NRO = 0;
}
```

No exemplo acima, a variável NRO é declarada duas vezes (uma vez em cada função) dentro do programa. Observe que o NRO da func1 não tem nenhuma relação com o NRO da func2, já que cada uma destas variáveis só são reconhecidas nos seus respectivos blocos de comando.

A maioria dos programadores declara todas as variáveis usadas por uma função imediatamente após o abre chaves da função. Porém, as **variáveis locais podem ser declaradas dentro de qualquer bloco de comando, <u>sempre após o abre chaves e antes de qualquer outro comando</u>**.

Exemplo:

```
void func()
{
        int NRO;
        scanf ("%d", &NRO);
        if (NRO = = 0)
        {
                char NOME[80];
                printf ("Entre com o nome:");
                gets(NOME);
        }
}
```

Contra-Exemplo:

```
void func()
{
        int NRO;
        NRO = 0;
        char NOME[80];
        gets(NOME);
}
/* Função com declaração incorreta, pois as variáveis devem ser declaradas no início do bloco de comandos, antes de qualquer outro comando */
```

Enquanto o trecho programa apresentado no exemplo acima funcionaria sem problemas, o trecho de programa descrito no contra-exemplo não seria executado na maioria dos compiladores em C.

Uma das vantagens no uso de variáveis locais é evitar efeitos colaterais indesejados, decorrentes de alterações acidentais no valor das variáveis fora da função. Outra vantagem é a redução no espaço de memória alocado pelo compilador para execução do programa.

PARÂMETROS FORMAIS

Se uma função usa argumentos de entrada, **ela deve declarar variáveis que receberão os valores desses argumentos**. Essas variáveis são denominadas ***parâmetros formais*** **da função** e se comportam como qualquer outra variável local da função.

Como mostrado no exemplo abaixo, as declarações dos parâmetros formais de uma função ocorrem depois de seu nome e dentro de parênteses:

Exemplo:

```
int is_in(char *texto, char letra)
{
        While (*texto)
        {
                if (*texto = = letra)
                        return 1;
                else
                        texto++;
        }
        return 0;
}
```

Os parâmetros formais declarados devem ser da mesma quantidade e tipo dos parâmetros utilizados na chamada da função. Se houver alguma discordância de tipo, resultados inesperados poderão ocorrer, uma vez que a linguagem C geralmente realiza alguma operação, mesmo em situações não usuais.

O padrão ANSI fornece os ***protótipos de funções***, que pode ser usado para ajudar a verificar se os argumentos usados para chamar a função são compatíveis com os parâmetros. Porém, para receber esse benefício extra, **os protótipos de funções devem ser incluídos explicitamente no programa**. Mais detalhes sobre protótipos de funções podem ser vistos no próximo capítulo.

VARIÁVEIS GLOBAIS

Variáveis globais são aquelas declaradas no início do programa, fora de qualquer função. Ao contrário das variáveis locais, este tipo de variável tem seus valores guardados durante toda a execução do programa, portanto, **são reconhecidas no programa inteiro e podem ser usadas por qualquer pedaço de código**.

O armazenamento de variáveis globais se encontra em uma região fixa da memória, separada para esse propósito pelo compilador C. **Variáveis globais são úteis quando o mesmo dado é usado em muitas funções em seu programa**. No entanto, deve-se evitar o uso desnecessário deste tipo de variável, uma vez que elas ocupam memória durante todo o tempo de execução do programa e não apenas quando são necessárias.

4.6. ESPECIFICADORES DE TIPO DE CLASSE DE ARMAZENAMENTO

Existem 4 especificadores de classe de armazenamento suportados pelo C: **extern**, **static**, **register** e **auto**. Esses especificadores são usados para informar ao compilador como a variável deve ser armazenada e deve preceder todo o resto da declaração, conforme a seguinte sintaxe:

especificador modificadores tipo_dado nome_variável

EXTERN

Como já visto, uma variável global só pode ser declarada uma única vez dentro de um arquivo. Assim, se declararmos em duplicidade uma variável dentro do programa, o compilador C pode apresentar uma mensagem de erro “nome de variável duplicado” ou, simplesmente, escolher uma das variáveis. O mesmo problema ocorre quando **declaramos variáveis globais em cada arquivo de um projeto**. Embora o compilador não emita nenhuma mensagem, uma vez que a compilação de cada arquivo é feita separadamente, estaríamos tentando criar duas ou mais cópias de uma mesma variável, e durante o processo de linkedição (que junta todos os arquivos do projeto) seria mostrada a mensagem de erro “rótulo duplicado”.

O modificador **extern** é utilizado para resolver este problema de duplicação. Isto é, declara-se as variáveis globais em um único arquivo, e nos demais repete-se a declaração da variável, precedida do modificador **extern**, como no exemplo:

```
/*Programa 1*/          /*Programa 2*/
int x, y;               extern int x, y;
char z;                 extern char z;
main(void)              main(void)
{                       {
}                       }
func1()                 Func2()
{                       {
x = 24;                 y = 33;
y = x+3;                x = y-10;
}                       }
```

STATIC

Variáveis *static* são variáveis permanentes. Isto é, apesar de não serem reconhecidas fora de suas funções ou arquivos, **mantém seus valores entre as chamadas**. Essa característica torna

este tipo de variável útil quando empregada em funções generalizadas e/ou de biblioteca, que podem ser utilizadas por outros programadores. O especificador ***static*** tem efeitos diferentes em variáveis locais e globais.

Quando é aplicado a **uma variável local**, o compilador cria **um armazenamento permanente**, quase da mesma forma que cria para variáveis globais. A diferença fundamental entre elas é que a variável local ***static*** é **reconhecida somente no bloco em que está declarada**.

No exemplo abaixo, a variável *NRO_SERIE* mantém seu valor entre as chamadas da função *obtem_nro_serie*. Isso significa que na primeira chamada será produzido o número de série 1010 e, a cada chamada da função, um novo número será criado com base no seu antecessor (ex: 1020, 1030, etc.).

Exemplo:

```
int obtem_nro_serie(void)
{
        static int NRO_SERIE = 1000;
        NRO_SERIE = NRO_SERIE + 10;
        return (NRO_SERIE);
}
```

Aplicar o especificador ***static*** **a uma variável global**, informa ao compilador para criar **uma variável que é reconhecida apenas no arquivo na qual a mesma foi declarada**. Isto é, embora a variável seja global, rotinas em outros arquivos não podem reconhecê-la ou alterar seu conteúdo diretamente.

AUTO

Por definição, as variáveis no C são da classe automática. O compilador automaticamente cria a variável cada vez que a função é chamada e elimina-a quando a função é finalizada.

O especificador ***auto*** instrui ao compilador para **classificar uma variável como automática**. O uso deste especificador é limitado, pois não se pode usá-lo com variáveis globais e as variáveis locais são automáticas por “default” (padrão).

REGISTER

O especificador *register* determina que “o acesso à variável seja o mais rápido possível”. Originalmente, quando este especificador era aplicado apenas à variáveis do tipo **int** e **char**, isto significada colocar o conteúdo destas variáveis nos registradores da CPU (daí vem o nome do especificador). Entretanto, como o padrão ANSI ampliou o uso deste especificador aos demais tipos de dados, tal tratamento foi modificado. No novo conceito do especificador, mantém-se o armazenamento original para os tipos **int** e **char** e, para os demais tipos (ex: matrizes e vetores), adota-se um tratamento diferenciado, de acordo com a implementação do compilador, visando tornar rápido o acesso ao seu conteúdo. Porém, nestes casos não ocorre um aumento substancial na velocidade de acesso.

Tal especificador só pode ser aplicado a variáveis locais e parâmetros formais, NÃO sendo permitido o seu uso com variáveis globais.

Além disso, como variáveis register podem ser armazenadas em um registrador da CPU, as mesmas não podem ter endereços. Portanto, este especificador impede o uso do operador **&**.

4.7. INICIALIZAÇÃO DE VARIÁVEIS

Na linguagem C, podemos inicializar uma variável junto com sua declaração. Para isto, basta colocar um sinal de igual e uma constante após o nome da variável, como no exemplo abaixo:

int NRO = 1000;

char LETRA = 'a', NOME[80] = "José";

Note que no exemplo acima a variável NRO será inicializada com 1000, a variável LETRA receberá o caracter 'a' e o vetor NOME receberá a string "José". Vale ainda ressaltar que, neste último caso, cada letra da string será associada a uma posição do vetor NOME, como ilustrado:

NOME[0] = 'J', NOME[1] = 'o', NOME[2] = 's', NOME[3] = 'é', NOME[4] = '\0'

Observe que, apesar da string de inicialização possuir apenas 4 caracteres, é inserido um caracter especial '**\0**' (***terminador nulo***) na quinta posição do vetor (*NOME[4]*). Este caracter deve ser utilizado nos testes condicionais das estruturas de controle (seleção ou repetição) para identificar o final da string, como no exemplo:

```
void main(void)
{
        char LETRA, FRASE[100];
        int I = 0, CONT = 0;
        printf("Digite uma frase:");
        scanf("%s\n\n",&FRASE);
        printf("Digite a letra que será contada:");
        scanf("%c\n\n",&LETRA);
        while ((FRASE[I] != '\0') && (I <= 100))
        {
                If (FRASE[I] = LETRA)
                {
                        CONT++; /* Equivalente ao comando CONT = CONT+1 */
                }
                ++I;
        }
        Printf("A letra %c apareceu %d vezes dentro da frase.\n", LETRA, CONT);
}
```

4.8. CONSTANTES

Constantes referem-se a valores fixos que o programa não pode alterar. No C, constantes podem ser de qualquer um dos cinco tipos de dados básicos e suas variações.

Constantes de caracter são delimitadas por apóstrofo ['] (ex: 'a', '%', '1'). Caracteres delimitados por aspas ["] são tratados como constantes string (cadeia de caracteres), portanto, ocupam um byte a mais na memória (ex: "a" = 'a' + '\0'). **Constantes inteiras são especificadas como números sem componentes fracionários** (ex: 69, -10). **Constantes reais podem ser expressas na forma decimal**, neste caso é requerido o ponto decimal **[.]** seguido da parte fracionária (ex: 3.141, -7.33); **ou através de notação científica** (ex: 4.34e-3).

Por default, o compilador C **encaixa uma constante numérica no menor tipo de dado compatível** que pode contê-la (ex: 10 ⇒ **int**, 60000 ⇒ **unsigned int**, 100000 ⇒ **log int**). A única exceção à esta regra são as **constantes reais**, as quais **são assumidas como double**. Apesar de esta associação padrão ser apropriada à maioria dos programas, o programador pode forçar a associação de uma constante a outro tipo de dado, através da utilização de um sufixo. Como por exemplo, -15L (força a utilização do tipo long int) e 3.141F (força a utilização do tipo float).

Muitas vezes, precisamos inserir constantes hexadecimais (base dezesseis) ou octais (base oito) no nosso programa. O C permite que se faça isto. **As constantes hexadecimais começam com 0x**. **As constantes octais começam em 0**.

Alguns exemplos:

```
0XEF        /* Constante Hexadecimal (8 bits) */

0x12A4      /* Constante Hexadecimal (16 bits) */

03212       /* Constante Octal (12 bits) */

034215432   /* Constante Octal (24 bits) */
```

5. COMANDO DE ATRIBUIÇÃO

O comando de atribuição do C é o sinal de igual (**=**). Ele é utilizado para atribuir à variável da esquerda o valor da variável, constante ou expressão contida à direita do sinal.

Sintaxe Geral:

Variável = expressão;

Sendo **expressão uma combinação de variáveis, constantes e operadores**.

5.1. OPERADORES

O C possui um grande conjunto de operadores que podem ser utilizados na manipulação de variáveis e constantes durante a execução do programa.

OPERADORES ARITMÉTICOS

Os operadores aritméticos são usados para desenvolver operações matemáticas. A lista dos operadores aritméticos do C é apresentada a seguir:

OPERADORES ARITMÉTICOS DO C

Operador	*Finalidade*
+	**Soma (inteira e ponto flutuante)**
-	**Subtração ou Troca de Sinal (inteira e ponto flutuante)**
*****	**Multiplicação (inteira e ponto flutuante)**
/	**Divisão (inteira e ponto flutuante)**
%	**Resto da Divisão (somente de inteiros)**

Uma particularidade do C refere-se ao fato do **valor retornado por uma expressão aritmética ser sempre do maior tipo de dado utilizado na expressão**. Assim, uma divisão de números

inteiros retornará a parte inteira do resultado (divisão inteira), enquanto a divisão de números do tipo float retornará um valor do mesmo tipo (divisão de números reais).

Todos os operadores descritos são binários (aplicáveis sobre duas variáveis), exceto pelo operador "-" quando aplicado para trocar o sinal de uma variável. Neste caso o operador simula a multiplicação da variável por -1.

Abaixo são apresentados alguns exemplos de utilização dos operadores aritméticos:

```
int A = 1, B = 2, C;

float X = 2.0, Y = 3.5, Z;

C = (2 * A + B) / B;     /* C receberá o valor 2 */

Z = X * Y + C;           /* Z receberá o valor 9.0 */

C = (A + B) % B;         /* C receberá o valor 1 */

C = -B;                  /* C receberá o valor -2 */
```

OPERADORES DE INCREMENTO E DECREMENTO

A linguagem C permite a utilização dos operadores unários "**++**" e "**--**" para incrementar (somar 1) ou decrementar (subtrair 1) o conteúdo de uma variável, respectivamente. Apesar destes operadores não serem aceitos em outras linguagens, eles possuem uma grande utilidade, principalmente na manipulação de variáveis contadoras. Abaixo são apresentados alguns exemplos de sua utilização:

```
++X;   /* Equivalente à expressão X = X + 1 */
X++;   /* Equivalente à expressão X = X + 1 */
--X;   /* Equivalente à expressão X = X - 1 */
X--;   /* Equivalente à expressão X = X - 1 */
```

Como pode ser notado, estes operadores podem ser pré-fixados ou pós-fixados. A diferença é que quando são pré-fixados, eles incrementam a variável antes de usar seu valor. Quando são pós-fixados, eles utilizam o valor da variável antes de incrementá-la. Para entender melhor esta diferença, veja os exemplos abaixo:

```
int X = 10, Y;
Y = X++;
/* Neste caso Y = 10 e X = 11 */
```

```
int X = 10, Y;
Y = ++X;
/* Neste caso Y e X = 11 */
```

EXPRESSÕES REDUZIDAS

As expressões reduzidas são aplicadas nos casos que uma mesma variável se encontra em ambos os lados do comando de atribuição. A forma geral destas expressões é:

variável operador = expressão;

EXEMPLO DE UTILIZAÇÃO DE EXPRESSÕES REDUZIDAS

Expressões Reduzidas	*Expressões Normais*
A += 5;	**A = A + 5;**
B - = 2.5;	**B = B – 2.5;**

Expressões Reduzidas	*Expressões Normais*
C *= 2;	**C = C * 2;**
D /= 7;	**D = D / 7;**
E %= 3;	**E = E % 3;**

Note que este tipo de expressão é muito útil para a manipulação de variáveis contadoras e/ou acumuladoras (que envolve somatórias, totais, etc.).

OPERADORES RELACIONAIS E LÓGICOS

Os operadores relacionais realizam ***comparações*** entre expressões, e seus resultados **sempre são valores lógicos** (falso – valores iguais a zero ou verdadeiro – valores diferentes de zero).

OPERADORES RELACIONAIS

Operador	*Finalidade*
==	**Igual**
>	**Maior**
<	**Menor**
>=	**Maior e igual**
<=	**Menor e igual**
!=	**Diferente**

Além dos operadores relacionais, a linguagem C ainda dispõe de operadores lógicos. Estes operadores podem ser utilizados para agrupar duas ou mais operações relacionais em um único teste condicional (operadores binários AND e OR) ou, ainda, modificar o resultado de uma operação relacional (operador unário NOT).

OPERADORES LÓGICOS

Operador	*Finalidade*
&&	**AND (E)**
||	**OR (OU)**
!	**NOT (NÃO)**

Abaixo é apresentado um programa exemplo que imprime os números pares entre 1 e 100. Observe neste exemplo que, quando o número é par, seu módulo é igual a 0. Portanto, utilizando o operador NOT, podemos converter este resultado para 1 (verdadeiro), disparando a cláusula então da estrutura de seleção IF (a qual será vista mais adiante).

```
# include <stdio.h>
void main(void)
{
        int NRO;
        for (NRO = 1; NRO <= 100; NRO++)
        {
                If ( !(NRO%2)) /* Verifica se o Nº é par ou ímpar */
```

```
                Printf("O número %d é par.", NRO);
        }
}
```

5.2. EXPRESSÕES CONDICIONAIS

O operador ternário "**?**" permite a atribuição condicional de valores a uma variável. Ele fornece uma forma alternativa de escrever uma tomada de decisão, através da seguinte sintaxe:

condição ? expressão1 : expressão2

Nesta estrutura, a condição é testada e, caso seja verdadeira, a expressão1 é assumida; senão é adotada a expressão2. Para exemplificar sua utilizado, segue abaixo um trecho de um programa que atribui à variável Z o maior valor entre as variáveis A e B:

```
Z = (A > B) ? A : B; /* Equivalente a Z = max(A,B) */
```

Outra forma de escrever este comando seria:

```
if (A > B)
        Z = A;
else
        Z = B;
```

5.3. CONVERSÃO ENTRE TIPOS DE DADOS

Quando uma expressão combina variáveis de tipos de dado diferentes, o compilador C verifica se as conversões entre estes tipos são possíveis. Caso não sejam, o processo de compilação é interrompido e uma mensagem de erro é exibida. Caso contrário, o compilador realiza todas as conversões necessárias, seguindo as seguintes regras:

1. Todos os **char** e **short int** são convertidos para **int**. Todos os **float** são convertidos para **double**.

2. Para pares de operandos de tipos diferentes: os de menor capacidade são convertidos para os de maior capacidade (ex: se um deles é **long double**, o outro também é convertido para **long double**).

Além disso, o programador pode **forçar que o resultado de uma expressão seja de um determinado tipo**. Para isto, ele deve utilizar operadores ***cast*** (modeladores). Sua forma geral é:

variável = (tipo_dado) expressão;

Abaixo é apresentado um exemplo da a utilização de operadores ***cast***:

```
# include <stdio.h>
void main(void)
{
        int NRO;
        float RESULTADO;
        scanf("%d", &NRO)¨;
        RESULTADO = (float) NRO/2;   /* Força a conversão de NRO em float */
        printf("\nA metade de %d é %f", NRO, RESULTADO);
}
```

Note que, se não tivéssemos utilizado o operador ***cast*** no exemplo acima, primeiro seria feito a divisão inteira de NRO por 2 e, depois, o resultado seria convertido para float.

5.4. PRECEDÊNCIA DE OPERADORES

Abaixo é apresentada a tabela de precedência dos operadores na linguagem C:

PRECEDÊNCIA DOS OPERADORES

Operadores	*Sentido de Execução*
( ) [ ]	**Esquerda p/ Direita**
++ -- - (unário) (cast)	**Direita p/ Esquerda**
*** / %**	**Esquerda p/ Direita**
+ - (binário)	
< > <= >=	
== !=	
!	
&&	
||	
?: (expressão condicional)	
= += *= /= - =	**Direita p/ Esquerda**

A seqüência das linhas indica a prioridade dos operadores, ou seja, os operadores localizados mais acima na tabela serão executados primeiro que aqueles descritos no final da tabela. Além disso, operadores descritos na **mesma linha possuem a mesma precedência**.

6. FUNÇÕES DISPONÍVEIS NO C

6.1. FUNÇÕES MATEMÁTICAS

Na tabela abaixo são apresentadas algumas das funções matemáticas disponíveis na linguagem C. Para utiliza-las é necessário declarar o arquivo de cabeçalho ***math.h***.

FUNÇÕES MATEMÁTICAS DO C

Função	*Sintaxe*	*Descrição*
ceil()	*ceil(REAL)*	Arredonda a variável REAL para cima (ex: ceil(3.4) = 4)
cos()	*cos(NRO)*	Calcula o cosseno de NRO (NRO em radianos)
exp()	*exp(NRO)*	Calcula ***e*** elevado à potência NRO
fabs()	*fabs(NRO)*	Calcula o valor absoluto de NRO
floor()	*floor(REAL)*	Arredonda a variável REAL para baixo (ex: floor(3.4) = 3)
log()	*log(NRO)*	Calcula o logaritmo natural de NRO
log10()	*log10(NRO)*	Calcula o logaritmo decimal de NRO

Função	Sintaxe	Descrição
pow()	*Pow(BASE,EXP)*	Calcula o valor de BASE elevado à potência EXP
sin()	*sin(NRO)*	Calcula o seno de NRO (NRO em radianos)
sqrt()	*sqrt(NRO)*	Calcula a raiz quadrada de NRO.
tan()	*tan(NRO)*	Calcula a tangente de NRO (NRO em radianos)

Além das operações de exponenciação, a função ***pow()*** também **realiza operações de radiciação**, uma vez que tal operação pode ser representada como uma exponenciação com o expoente fracionário.

exemplo: $\sqrt[3]{X} = X^{\frac{1}{3}}$

6.2. FUNÇÕES TOLOWER() E TOUPPER()

As funções ***tolower()*** e ***toupper()*** estão definidas no arquivo de cabeçalho ***ctype.h*** e têm por finalidade, **converter uma letra maiúscula em minúscula e vice-versa**, respectivamente. Suas sintaxes são:

variável_char = tolower(caracter);

variável_char = toupper(caracter);

Qualquer caracter diferente de letra é retornado sem modificações.

A seguir é apresentado um exemplo de utilização destas funções:

```
#include <stdio.h>
#include <ctype.h>
void main(void)
{
        char LETRA;
        LETRA = getchar();
        putchar( tolower(LETRA));
        putchar( toupper(LETRA));
}
```

6.3. FUNÇÃO CLRSCR()

A função ***clrscr()*** é utilizada para limpar todo o conteúdo da tela e posicionar o cursor no seu início, ou seja, canto superior esquerdo da tela (linha 1 , coluna 1). Sua forma geral é:

clrscr();

Para utilizar esta função é necessário declarar o arquivo de cabeçalho ***conio.h***, conforme apresentado no exemplo abaixo:

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
```

```
        printf("Esta é a primeira linha\n");
        printf("Esta é a segunda linha\n");
        printf("Esta é a terceira linha\n");
        clrscr();
        printf("Esta é a nova primeira linha\n");
        printf("Esta é a nova segunda linha\n");
}
```

6.4. FUNÇÃO GOTOXY()

A função ***gotoxy()*** é utilizada para posicionar o cursor na posição (coordenadas de linha e coluna) desejada. Sua forma geral é:

gotoxy(coluna, linha);

Para utilizar esta função é necessário declarar o arquivo cabeçalho ***conio.h***, conforme apresentado no exemplo abaixo:

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
        int LINHA, COLUNA;
        char TEXTO[10];
        printf("Digite a linha e coluna onde deseja escrever o texto:");
        scanf("%d,%d", &LINHA, &COLUNA);
        printf("\nDigite o texto a ser mostrado:");
        gets(TEXTO);
        clrscr();
        gotoxy(COLUNA, LINHA);
        puts(TEXTO);
}
```

7. COMANDOS DE ENTRADA E SAÍDA

7.1. PRINTF()

A função ***printf()*** possibilita a saída de valores (constantes, variáveis ou resultados de expressões). Esta função converte e imprime seus argumentos na saída padrão seguindo o formato especificado na ***string de controle***. Sua sintaxe geral é:

printf("string de controle", lista de argumentos);

A ***string de controle***, delimitada por aspas, descreve tudo que a função irá colocar na tela. Esta string não mostra apenas os caracteres que devem serão apresentados, mas também os respectivos formatos e posições das constantes, variáveis e expressões listadas na ***lista de argumentos***. Assim, resumidamente, podemos dizer que a ***string de controle*** consiste de três tipos de caracteres:

- **Caracteres diversos que serão impressos** na tela (correspondem ao texto que será exibido);

- **Caracteres especiais de escape** "****", utilizados para auxiliar na movimentação do cursor na tela ou do carro de impressão (alguns destes caracteres serão vistos na seção 7.7); e
- **Caracteres especificadores de formato "%"**, que definem a posição e a maneira como cada uma das variáveis contidas na ***lista de argumentos*** serão exibidas (tipo e tamanho do dado).

ALGUNS ESPECIFICADORES DE FORMATO DA FUNÇÃO PRINTF()

Especificador	Tipo do Argumento	Descrição
%d	int	Valor inteiro decimal
%i	int	Valor inteiro decimal
%o	int	Valor inteiro octal
%x	int	Valor inteiro hexadecimal
%X	int	Valor inteiro hexadecimal
%u	int	Valor inteiro decimal sem sinal (*unsigned int*)
%c	int	Um caracter em formato ASCII (código binário correspondente)
%s	char	Uma cadeia de caracteres (*string*) terminada em "**\0**"
%f	float	Valor em ponto flutuante no formato [-]m.dddddd, onde o Nº de d's é dado pela precisão (padrão é 6)
%e	float ou double	Valor em ponto flutuante em notação exponencial no formato [-]m.ddddd e±xx, onde o Nº de d's é dado pela precisão
%E	float ou double	Valor em ponto flutuante em notação exponencial no formato [-]m.ddddd E±xx, onde o Nº de d's é dado pela precisão
%g	float ou double	Valor em ponto flutuante no formato %e (quando o expoente for menor que -4 ou igual a precisão) ou %f (nos demais casos). Zeros adicionais e um ponto decimal final não são impressos
%G	float ou double	Valor em ponto flutuante no formato %E (quando o expoente for menor que -4 ou igual a precisão) ou %f (nos demais casos). Zeros adicionais e um ponto decimal final não são impressos
%%	-	Exibe o caracter '%'

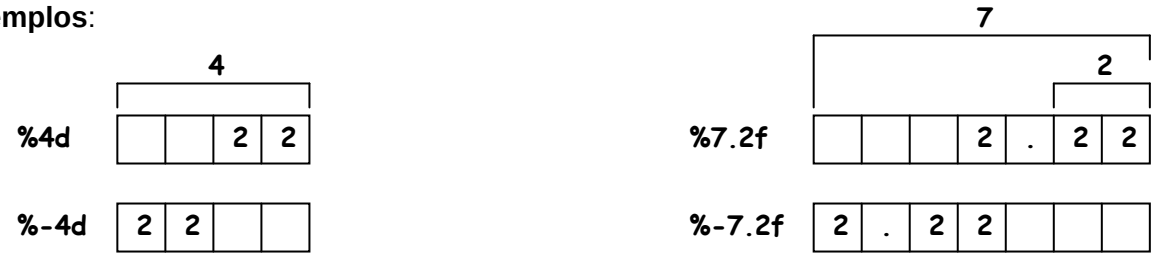
Entre o caracter "%" e o caracter especificador do tipo de dado pode vir as seguintes alternativas:

- **Um sinal de menos** que determina o ajustamento/alinhamento à esquerda do argumento convertido;
- **Um número** que especifica a largura (tamanho) mínima do campo que receberá o argumento convertido. Se necessário, ele será preenchido com espaço à esquerda (ou à direita, se o ajustamento à esquerda for solicitado) para compor a largura do campo;
- **Um ponto** que separa a largura do campo da sua precisão (Nº de casas decimais);

- **Um número ou a precisão**, que especificam o número máximo de caracteres a ser impresso em uma *string*, ou o número de dígitos após o ponto decimal de um valor real, ou o número mínimo de dígitos para um inteiro;

- Um "**h**" ou "**l**" se o número inteiro tiver que ser impresso como ***short*** ou ***long***, respectivamente.

Exemplos:



Um programa exemplo ilustrando o uso destes especificadores de formato na função ***printf()*** é apresentado a seguir:

```
#include <stdio.h>
void main(void)
{
    short int CH = 47;
    printf("%-10.2f %-10.2f %-10.2f \n", 8.0, 15.253, 584.13);
    printf("%04d \n", 21);
    printf("%6d \n", 21);
    printf("%10.3f \n", 3456.78);
    printf("%3.1f \n", 3456.78);
    printf("%hd %c \n", CH);
}
```

Este programa exemplo reproduzirá as seguintes saídas:

```
8.00      15.25      584.13          /* Alinhamento à esquerda (espaços à direita)*/
0021                                 /*O Nº é complementado com zeros à esquerda*/
  21                                 /*O Nº é complementado com espaços à esquerda */
 3456.780
3456.8                               /* A parte fracionada é arredondada para 1 casa */
47 /                                 /* 47 é o código ASCII do caracter '/' */
```

7.2. SCANF()

A função ***scanf()*** lê caracteres da entrada padrão (teclado), interpreta-os segundo a especificação de formato e armazena os resultados em variáveis declaradas no programa. Sua forma geral é:

scanf ("string de controle", lista de endereços de memória das variáveis);

A representação do endereço de memória das variáveis que receberão os dados de entrada é fornecido **pelo operador "&", seguido pelo nome da variável**. Além disso, assim como na função ***printf()***, esta função também utiliza os caracteres especiais de controle via códigos de barra invertida (veja seção 7.7) e de formatação "**%**" (conforme apresentado na tabela abaixo).

ALGUNS ESPECIFICADORES DE FORMATO DA FUNÇÃO SCANF()

Especificador	Descrição
%d	Indica um valor inteiro decimal
%i	Indica um valor inteiro (podendo estar na base decimal, octal com inicial **0** (zero) ou hexadecimal com inicial **0X**
%u	Indica um valor inteiro decimal sem sinal
%c	Indica um caracter (*char*)
%s	Indica uma *string*
%f, %e, %g	Indica um valor em ponto flutuante de precisão simples (*float*)
%lf, %le, %lg	Indica um valor em ponto flutuante de precisão dupla (*double*)
%o	Indica um valor inteiro octal (com ou sem zero inicial)
%x ou %X	Indica um valor inteiro hexadecimal (com ou sem "0x" inicial)

No exemplo abaixo, a função ***scanf()*** lerá o número digitado como ***float***, armazenando-o no endereço da variável ***NRO***. Os números digitados serão somados até que seja digitado ***CTRL-Z***. Como saída serão apresentados a soma parcial (subtotal) a cada dez números lidos e o total geral ao final do programa.

```
# include <stdio.h>
void main(void)
{
    float SOMA, NRO;
    int CONT;
    CONT = SOMA = 0;
    printf ("\n Entre com os números a serem somados ou digite CTRL-Z para terminar |n");
    while (scanf ("%f", &NRO) == 1)
    {
            CONT++;
            if (CONT == 10)
            {
                    printf("\t Subtotal: %.2f \n", SOMA += NRO);
                    CONT = 0;
            }
            else
```

```
                SOMA += NRO;
        }
        printf ("\t Total: %.2f \n", SOMA);
    }
```

7.3. GETCHAR()

A macro ***getchar()*** retorna o próximo caracter de entrada toda vez que é chamada, ou **EOF** (End Of File) quando for fim de arquivo. Esta macro **interpreta o caracter de entrada pelo seu valor ASCII**, ou seja, **seu retorno é um número inteiro**, correspondente ao código binário do caracter lido. Ela usa uma técnica chamada de **entrada "bufferizada"**, ou seja, os caracteres digitados pelo usuário não estão disponíveis para a macro ***getchar()*** até que seja pressionada a tecla **<ENTER>**.

A forma básica de utilização desta macro é:

variável = getchar();

No programa exemplo abaixo, será impresso o código ASCII correspondente ao caracter digitado.

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
        int NUM;
        clrscr();
        printf ("\nEntre com um caracter ou tecle espaço para terminar: ");
        while ((NUM = getchar()) != ' ')
        {
                if (NUM != 10)
                {
                        printf("\n\tCaracter\tCódigo ASCII\n\n\t %c \t %d\n", NUM, NUM);
                        printf("\nEntre com um caracter ou tecle espaço para terminar: ");
                }
        }
getchar();
}
```

Neste exemplo, a linha "*getchar();*" é utilizada para manter a exibição da tela do DOS até que um caracter qualquer seja pressionado.

*Obs: o arquivo de cabeçalho **stdio.h** contém a declaração do identificador **EOF*** (***#define EOF -1***).

7.4. GETCH() e GETCHE()

As funções ***getch()*** e ***getche()*** retornam o caracter pressionado. A função ***getche()*** imprime o caracter na tela antes de retorná-lo, enquanto que a função ***getch()*** apenas retorna o caracter sem imprimi-lo. Ambas são definidas no arquivo de cabeçalho ***conio.h***, portanto, não pertencem ao padrão ANSI. A sintaxe destas funções é similar a da macro ***getchar()***.

7.5. GETS()

A função ***gets()*** lê uma **string** e coloca-a no endereço apontado pelo seu argumento ponteiro para caractere. Para efeito prático, fará parte da string tudo o que for digitado até o retorno do carro (tecla <ENTER>), sendo este substituído pelo terminador nulo "**\0**". Sua sintaxe geral é:

gets(nome_variavel);

Se for ultrapassado o espaço reservado para a string, esta função sobrepõe os valores na memória, podendo ocasionar um erro grave.

A seguir é apresentado um programa exemplo da função ***gets()***.

```
# include <stdio.h>
void main(void)
{
    char MSG[21];
    printf("\n\nDigite uma mensagem com no máximo 20 caracteres: ");
    gets(MSG);
    printf("\n\nA mensagem digitada é: %s", msg);
    getchar();
}
```

7.6. PUTS()

A função ***puts()*** apresenta na tela a string passada como argumento, seguida de uma nova linha (**\n**). Ela retorna um valor não negativo se houver sucesso no processo ou, caso contrário, retorna **EOF**. Sua forma geral é:

puts(string);

Esta função reconhece os mesmos caracteres especiais de escape (**backslash – "\"**) que a função ***printf()***. Entretanto, uma chamada à função ***puts()*** requer menos esforço de processamento que a mesma chamada para a função ***printf()***, uma vez que a primeira pode enviar somente strings, enquanto que a segunda, além de strings, pode enviar outros tipos de dados, bem como fazer conversão de formatos. Portanto, a função ***puts()*** ocupa menos espaço de memória e é executada mais rapidamente que a função ***printf()***. A seguir é apresentado um exemplo de sua utilização.

```
#include <stdio.h>
void main(void)
{
        char STRING[10];
        puts("Exemplo de utilização da função puts.\n Entre com uma string:");
        gets(STRING);
        puts("\n A string digitada foi:");
        puts(STRING);
}
```

7.7. PUTCHAR()

A macro ***putchar()*** coloca um único caracter na saída padrão. Sua sintaxe geral é:

putchar(variável);

A variável passada como argumento para a macro pode ser do tipo ***int*** (contendo o código binário do caracter desejado) ou ***char***.

Além disso, assim como a função ***puts()***, ela retorna **EOF** em caso de erro.

No exemplo a seguir, a macro é utilizada para apresentar na saída padrão (que por "default" é a tela) os caracteres digitados pelo usuário em letras minúsculas.

```
#include <stdio.h>
#include <ctype.h>
void main(void)
{
    char C;
    puts("Entre com os caracteres desejados ou tecle CTRL-Z para terminar:\n");
    while ((C = getchar()) != EOF)
            putchar (tolower(C));
}
```

7.8. CARACTERES DE ESCAPE

Para nos facilitar a tarefa de programar, a linguagem C disponibiliza vários códigos chamados **códigos de barra invertida - *backslash* ou caracteres de escape**. Uma lista com os principais códigos é dada a seguir:

CÓDIGOS DE BARRA INVERTIDA

Código	Significado
\a	Sinal sonoro ("beep")
\b	Retrocesso ("backspace")
\n	Mudança de linha ("new line")
\f	Avanço de página ("form feed")
\r	Retorno do carro da impressora
\t	Tabulação horizontal ("tab")
\v	Tabulação vertical
\0	Nulo (0 em decimal)
****	Impressão da barra invertida
\"	Impressão das aspas
\'	Impressão do apóstrofo
\?	Impressão do ponto de interrogação
\N	Constante octal (N é o valor ASCII em octal)
\xN	Constante hexadecimal (N é o valor ASCII em hexa)

Estes caracteres de controle podem ser usados como qualquer outro dentro das ***strings de controle*** das funções ***printf()*** e ***scanf()***.

8. ESTRUTURAS DE SELEÇÃO OU DECISÃO

Este tipo de estrutura permite direcionar o fluxo lógico do programa para blocos distintos de instruções, conforme uma condição de controle (normalmente avaliação de expressões lógicas e/ou relacionais).

8.1. COMANDO IF

O comando ***if*** é o principal comando para codificar a tomada de decisão em C. Basicamente, ele examina a condição especificada e, **se a condição for verdadeira** (diferente de zero), o programa **executa o bloco de comandos** que segue; caso contrário, ou seja, **se a condição for falsa** (igual a zero), **o bloco de comandos não será executado**.

A sintaxe geral do comando ***if*** é a seguinte:

```
If (condição)
{
        bloco de comandos;
}
```

O bloco de comandos deve ser delimitado por uma chave aberta e uma fechada. No entanto, se o bloco é composto por **apenas um único comando**, **as chaves se tornam opcionais** e podem ser omitidas.

Um exemplo de utilização deste comando é apresentado a seguir:

```
#include <stdio.h>
int main (void)
{
        int a,b;
        printf("Digite dois números inteiros: ");
        scanf("%d %d", &a, &b);
        if ((a < 0) && (b < 0))
        {
                printf("\nForam digitados 2 números negativos.");
        }
        return 0;
}
```

8.1.1. CLÁUSULA ELSE

A inclusão da cláusula ***else*** permite que um segundo bloco de comandos seja executado, caso a condição testada pelo comando ***if*** seja falsa (igual a zero). Sua forma geral é:

```
If (condição)
{
        bloco de comandos 1;  /* Executado quando a condição for verdadeira */
}
else
{
        bloco de comandos 2;  /* Executado quando a condição for falsa */
```

```
}
```

A seguir é apresentado um exemplo da utilização deste comando. Este programa tem como entrada um número inteiro e sua saída é uma frase informando se o número fornecido é par ou ímpar.

```
#include <stdio.h>
void main (void)
{
        int NRO;
        printf("Digite um número inteiro: ");
        scanf("%d", &NRO);
        if (NRO%2 == 0)
                printf("\nO número é par");
        else
                printf("\nO número é ímpar");
}
```

Neste programa não foram utilizadas as chaves para delimitar os blocos do comando ***if-else***. Além disso, observe que os blocos de comandos estão endentados (recuo de linha) à estrutura a qual pertencem. Como já visto em algoritmos, esta **endentação** não é obrigatória, mas **é fundamental para uma maior clareza do código**, uma vez que permite identificar visualmente o início e fim de cada bloco. Todavia, **nem sempre o uso da endentação aliado à remoção das chaves facilita a associação dos blocos de comandos a suas respectivas estruturas**, como apresentado no contra-exemplo abaixo:

```
#include <stdio.h>
void main (void)
{
        int TEMP;
        printf("Digite a temperatura: ");
        scanf("%d", &TEMP);
        if (TEMP < 30)
                if (TEMP > 20)
                        printf("\nTemperatura agradável");
        else
                printf("\nTemperatura muito quente");
}
```

A idéia desse programa era imprimir a mensagem "Temperatura agradável" se fosse fornecido um valor entre 20 e 30 e imprimir a mensagem "Temperatura muito quente" se fosse fornecido um valor maior que 30. Entretanto, neste caso, a segunda mensagem será exibida quando a temperatura for menor ou igual a 20. Isto porque, na linguagem C, **um *else* é associado ao último *if* que não tiver seu próprio *else***. Assim, nos casos onde tal associação não é apropriada, **deve-se criar explicitamente os blocos de comando através do emprego das chaves**. Abaixo é apresentado o programa com as alterações necessárias ao bom funcionamento:

```
#include <stdio.h>
void main (void)
{
        int TEMP;
```

```
        printf("Digite a temperatura: ");
        scanf("%d", &TEMP);
        if (TEMP < 30)
        {
                if (TEMP > 20)
                        printf("\nTemperatura agradável");
        }
        else
                printf("\nTemperatura muito quente");
}
```

8.1.2. NINHOS DE IF

Na linguagem C, assim como já apresentado em algoritmos, é possível utilizar-se de aninhamentos de comandos ***if***. Isto é, um comando ***if*** pode conter outros comandos ***if's*** dentro de seus blocos de comandos.

Exemplo:

```
#include <stdio.h>
void main(void)
{
        int TEMP;
        printf("Digite a temperatura: ");
        scanf("&d", &TEMP);
        if (TEMP < 20)
        {
                if (TEMP < 10)
                        printf("\nTemperatura muito fria.");
                else
                        printf("\nTemperatura fria.");
        }
        else
        {
                 if (TEMP < 30)
                        printf("\nTemperatura agradável.");
                else
                        printf("\nTemperatura quente.");
        }
}
```

8.1.3. CLÁUSULA ELSE IF

A cláusula ***else if*** é usada para escrever decisões com escolhas múltiplas (teste de várias condições). Funciona de forma similar ao **aninhamento de comandos *if* dentro do blocos de comandos das cláusulas *else* dos *if's* superiores**. Sua sintaxe geral é:

```
If (condição1)
{
```

```
        bloco de comandos 1;  /* Executado quando a condição1 for verdadeira */
}
else if (condição2)
{
        bloco de comandos 2;  /* Executado quando a condição2 for verdadeira */
}
else if (condição3)
{
        bloco de comandos 3;  /* Executado quando a condição3 for verdadeira */
}
•
•
•
else if (condiçãoN)
{
        bloco de comandos N;  /* Executado quando a condiçãoN for verdadeira */
}
else
{
        bloco de comandos (N+1);   /* Executado quando todas as condições
                                      anteriores forem falsas */
}
```

Nesta estrutura, as expressões condicionais são avaliadas em ordem e, se alguma delas for verdadeira, o bloco de comandos correspondente será executado e o encadeamento será finalizado (nenhuma outra condição da estrutura será testada).

Abaixo é apresentado uma modificação do programa anterior, contendo a cláusula ***else if***:

```
#include <stdio.h>
void main(void)
{
        int TEMP;
        printf("Digite a temperatura: ");
        scanf("&d", &TEMP);
        if (TEMP < 10)
                printf("\nTemperatura muito fria.");
        else if (TEMP < 20)
                printf("\nTemperatura fria.");
        else if (TEMP < 30)
                printf("\nTemperatura agradável.");
        else
                printf("\nTemperatura quente.");
}
```

8.2. COMANDO SWITCH

Em um comando ***switch***, o computador **compara uma variável sucessivamente contra uma lista de constantes**. Caso uma das constantes descrita nas cláusulas ***case*** seja igual ao valor

atual da variável, o computador executa o comando ou bloco de comandos que estejam associados àquela opção.

Este tipo de estrutura difere do ***if***, pois **só pode testar igualdade**, enquanto o ***if*** pode testar qualquer tipo de expressão lógica.

A sua sintaxe geral é:

```
switch (variável_testada)
{
case valor_1: {
                bloco-comandos;
                break;
                }
case valor_2: {
                bloco-comandos;
                break;
                }
●
●
●
case valor_N: {
                bloco-comandos;
                break;
                }
default: {
                bloco-comandos;
        }
}
```

O comando ***break*** **é usado para terminar uma seqüência de comandos**. Tecnicamente **o seu uso é opcional**. Entretanto, quando omitido, a execução continuará nos comandos do próximo ***case*** até que o computador encontre um ***break*** ou o fim do comando ***switch - case***.

A cláusula *default* também é opcional e serve para tratar os casos que não se enquadram nas condições testadas nas cláusulas *case*. Se não for utilizado, nenhum comando será executado caso todas as opções ***case*** falhem.

O comando ***switch*** é freqüentemente utilizado na construção de menus, como ilustrado no exemplo a seguir:

```
/* construcao de menu utilizando o comando switch - case */

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

main()
{

 float x,y,r;
 char op;

 clrscr();
 printf ("Entre com o 1º valor: ");
 scanf("%f", &x);
 printf ("Entre com o 2º valor: ");
 scanf("%f", &y);
 op = '0';
 while (op !='5')
```

```
        {
           clrscr();
           printf ("Menu Principal\n");
           printf ("1. Soma \n");
           printf ("2. Subtracao \n");
           printf ("3. Multiplicacao \n");
           printf ("4. Divisao\n");
           printf ("5. Sair\n");
           printf ("Escolha uma das opções acima:");
           op = getche();
           clrscr();
           if (op!='5')
            {
            switch (op)
             {
               case '1': {
                          printf (" Calculo da Soma \n");
                          r = x+y;
                          printf (" %.1f   +  %.1f  =   %.1f ", x,y,r);
                          getch();
                          break;
                         }
               case '2': {
                          printf (" Calculo da Subtracao \n");
                          r = x-y;
                          printf (" %.1f   -  %.1f  =   %.1f ", x,y,r);
                          getch();
                          break;
                        }
               case '3': {
                          printf (" Calculo da Multiplicacao \n");
                           r = x*y;
                          printf (" %.1f   x  %.1f  =   %.1f ", x,y,r);
                          getch();
                          break;
                        }
               case '4': {
                           printf (" Calculo da Divisao \n");
                           if (y != 0)
                              {
                               r = x/y;
                               printf (" %.1f   /  %.1f  =   %.1f ", x,y,r);
                              }
                             else printf(" Divisao por zero!");
                          getch();
                          break;
                         }
               default:{
                        printf("Opção inválida!");
                        getch();
                       }
             } /* fim do switch */
            }  /* fim do if */

        } /* fim do while */
      }
```

9. ESTRUTURAS DE REPETIÇÃO

Permite executar repetidamente um bloco de instruções ate que uma condição de controle seja satisfeita.

9.1. COMANDO WHILE

O comando ***while*** executa um bloco de comandos enquanto a condição testada for verdadeira (diferente de zero). Sua sintaxe geral é:

```
while (condição de teste)
{
        boco_comandos;
}
```

Este comando **verifica a condição de teste no início do *"loop"***. Isto significa que **o seu bloco de comandos poderá ou não ser executado**, de acordo com o resultado desta verificação.

Abaixo é apresentado um programa exemplo deste comando:

```
# include <stdio.h>
void main(void)
{
        int i = 1, soma = 0;
        while (i <= 10)
        {
                soma += i;
                i++;
        }
        printf("A soma dos números inteiros de 1 a 10 é %d", soma);
        getchar();
}
```

Caso a condição testada seja um número ou uma expressão matemática, o *"loop"* será executado **até que o valor do número ou da expressão seja igual a zero**.

Assim como no comando ***if***, se o bloco de comando for formado por uma única instrução, não é necessário utilizar as chaves.

9.2. COMANDO DO – WHILE

O comando ***do – while*** é similar ao comando ***while***, exceto pelo fato que, ele **primeiro executa o bloco de comandos para depois verificar a condição de teste**, ou seja, neste comando, as instruções contidas no *"loop"* são executadas pelo menos uma vez. Além disso, se a condição testada for verdadeira, **o bloco de comandos será novamente executado até que a condição de teste se torne falsa**. Sua sintaxe geral é:

```
do
{
        bloco_comandos;
} while (condição de teste);
```

Abaixo é apresentado um exemplo de utilização deste comando:

```
# include <stdio.h>
void main(void)
{
```

```
        int i = 1, soma = 0;
        do
        {
                soma += i;
                i++;
        } while (i <= 10)
        printf("A soma dos números inteiros de 1 a 10 é %d", soma);
        getchar();
}
```

O uso deste comando é indicado para os casos onde um conjunto de instruções precisa ser executado pelo menos uma vez e, então, repetido condicionalmente.

9.3. COMANDO FOR

O comando ***for*** é utilizado quando se deseja executar instruções repetidas que envolvam o incremento/decremento das **variáveis de controle do *"loop"*** (ex: leitura e/ou manipulação de vetores e matrizes). Sua sintaxe geral é:

```
for (inicialização ; condição ; incremento)
{
        bloco_comandos;
}
```

Onde:

- **Inicialização** é geralmente um comando de atribuição utilizado para determinar a variável de controle do *"loop"* e seu valor inicial;
- **Condição** é uma expressão relacional que determina o critério de parada do comando ***for***;
- **Incremento** é uma expressão aritmética que define como a variável de controle se altera a cada repetição do comando ***for***.

Estas três seções devem ser **separadas por ponto e vírgula** (**;**).

A seqüência de operação do comando ***for*** é a seguinte:

- **Inicialização da variável** de controle;
- **Teste da condição**. Se o resultado for diferente de zero (verdadeiro) executa-se o bloco de comandos;
- **Incremento da variável** de controle.

O comando ***for*** continua a execução enquanto o resultado do teste for verdadeiro. Assim que a condição for falsa (igual a zero), a execução do programa será desviada para os comandos subseqüentes ao comando ***for***. Um exemplo deste comando é apresentado abaixo:

```
# include <stdio.h>
void main(void)
{
        int i, soma=0;
```

```
        for (i=1; i <= 10; ++i)
                soma += i;
        printf("A soma dos números inteiros de 1 a 10 é %d", soma);
        getchar();
}
```

VARIAÇÕES DO COMANDO

Qualquer uma das três instruções que formam o comando *for* pode ser omitida, embora os ponto e vírgulas devam permanecer.

Se o teste for omitido, o *"loop"* se tornará infinito, podendo ser interrompido caso seja usado um comando de desvio (***break***, ***return*** ou ***goto***) no bloco de comandos do ***for***. Veja o exemplo:

```
# include <stdio.h>
void main(void)
{
        char x;
        printf("\nDigite quaisquer caracteres ou tecle A para sair:\n");
        for (;;)
        {
                x=getchar();    /* Lê um caracter */
                if (x == 'A')
                        break; /* Sai do loop */
        }
        printf("\nVocê saiu do Loop");
        getchar();
}
```

Quando a parte do incremento é omitida, não é feita nenhuma alteração na variável de controle. Assim, ou o *"loop"* será executado infinitamente, ou esta alteração deverá ser feita dentro do bloco de comandos do ***for***, como no seguinte exemplo:

```
# include <stdio.h>
void main(void)
{
        int x;
        printf("\nDigite quaisquer numeros ou 123 para sair:\n");
        for (x=0; x != 123;)
                scanf ("%d", &x);       /* Lê um número */
        printf("\nVocê saiu do Loop");
        getchar();
}
```

O comando ***for*** permite que haja **argumentos múltiplos nas três seções**. Neste caso, **os argumentos adicionais deverão ser separados por vírgula** (**,**), como ilustrado no exemplo:

```
# include <stdio.h>
void main(void)
{
        int x, y;
```

```
        for (x=0, y=100; x < y; x++, --y)
                printf ("x = %d e y = %d\n", x, y);
        printf("Estes foram os valores de x e y enquanto x < y");
        getchar();
}
```

Por fim, tanto a **inicialização** quanto o **incremento** podem estar **fora dos parênteses de controle** do ***for***, como no exemplo:

```
# include <stdio.h>
void main(void)
{
        int x = 0;
        for (; x < 10 ;)
        {
                printf ("x = %d \n", x);
                x++;
        }
        getchar();
}
```

10. COMANDOS DE DESVIO

Às vezes, é conveniente se controlar a saída de uma estrutura de repetição (***for***, ***while*** ou ***do - while***), de outro modo além dos testes condicionais de início ou fim do mesmo. Para isto pode-se usar alguns comandos que permitem desviar a execução normal de um programa. Tais comandos são apresentados a seguir.

10.1. COMANDO BREAK

O comando ***break*** encerra o processamento de estruturas de repetição ou a execução de um comando ***switch***.

O exemplo a seguir mostra um programa que faz uso do comando ***break***.

```
#include <stdio.h>
void main(void)
{
        int NRO, X;
        char TEXTO[80];
        printf("Digite um número inteiro ou zero para sair: ");
        scanf("%d", &NRO);
        while (NRO != 0)
        {
                TEXTO = "O número é primo";
                for (X = 2; X < NRO; X++)
                {
                        if (NRO%X == 0)
                        {
                                TEXTO = "O número não é primo";
```

```
                        break;
                    }
                }
                puts(TEXTO);
                scanf("%d", &NRO);
            }
        }
```

Quando executado dentro de uma repetição aninhada (uma estrutura de repetição dentro de outra), **o comando interrompe a repetição mais interna**, continuando o processamento na próxima estrutura de repetição mais interna (estrutura superior imediata).

10.2. COMANDO RETURN

O comando ***return*** **é o mecanismo de saída de uma função**, **retornando um valor para a função que a chamou**. Sua sintaxe é:

return (expressão_de_retorno);

Onde, ***expressão_de_retorno*** corresponde ao **valor que será utilizado no lugar da chamada da função**.

Se o comando ***return*** estiver dentro de uma estrutura de repetição, ele não só **sairá da estrutura como também da função** onde foi executado. O exemplo abaixo mostra o ***return*** dentro da função que eleva um número ao quadrado.

```
# include <stdio.h>

double quadrado(double y);
void main(void)
{
        double x;
        printf ("Digite um número: \n");
        scanf ("%lf", &x);
        x = quadrado(x);
        printf ("O quadrado do número é: %lf", x);
        getchar();
}
double quadrado (double y)
{
        return (y *= y);
}
```

O uso dos parênteses em torno da ***expressão_de_retorno*** **é opcional**.

10.3. COMANDO CONTINUE

O comando ***continue*** pára a seqüência de instruções que está sendo executada (iteração atual), passando para a próxima iteração, ou seja, voltando para a 1ª linha do bloco de comandos do *"loop"*, conforme apresentado no programa-exemplo a seguir.

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

```
void main(void)
{
        int x;
        printf ("Digite um número inteiro positivo ou <100> para sair: \n");
        do
        {
                scanf ("%d", &x);
                if (x < 0)
                {
                        printf ("O número digitado não será impresso por ser negativo. \n");
                        continue;
                }
                printf("O número digitado foi %d. \n", x);
        } while (x != 100);
        printf ("O número 100 foi digitado.");
}
```

Neste exemplo, o número digitado só será impresso na tela se for um número positivo. Para isto, foi empregado o comando ***continue***, o qual provoca a interrupção do bloco de comandos e o retorno ao comando ***scanf()***.

10.4. COMANDO GOTO

O comando ***goto*** **desvia a seqüência de execução lógica de um programa**, ou seja, ele passa o controle de programa (realiza um salto incondicional) para a linha de comando identificada por um rótulo ("***label***"). O rótulo pode ser qualquer nome que atenda às regras de criação de identificadores e **deve ser declarado na posição para onde se deseja saltar, seguido de dois pontos** (**:**). **Tanto o comando *goto* quanto o seu rótulo devem estar declarados dentro da mesma função**.

A sintaxe geral do comando é:

```
goto nome_rotulo
•
•
•
nome_rotulo:
  bloco_comandos;
```

Este comando não é necessário, podendo sempre ser substituído por outras estruturas de controle. Inclusive, **o seu uso não é recomendável**, pois **pode tornar o programa ilegível e confuso**, principalmente se existirem várias ocorrências do comando em diferentes partes do programa.

Existem algumas situações muito específicas onde este comando pode tornar o código mais fácil de entender. Um destes casos é quando utilizado para abandonar o processamento de uma estrutura altamente aninhada (vários *loops* e *ifs*), onde o uso do comando ***goto*** é mais elegante e rápido que a utilização de vários comandos ***break***.

No exemplo apresentado abaixo, se for digitado o número **3**, o controle do programa será desviado para o ***label*** **saída**.

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

```
void main(void)
{
        int x = 0;
        printf ("Entre com caracteres ou tecle 3 para sair: \n");
        for (;;)
        {
                x = getchar(); /* Lê um caracter */
                if (x == '3')
                        goto saida;  /* Sai do Loop */
                else
                        putchar(x);
        }
        saida:
        printf("Você saiu do Loop");
}
```

10.5. FUNÇÃO EXIT()

A função ***exit()*** é encontrada na biblioteca padrão do C. Sua utilização é limitada, pois **provoca o encerramento imediato de um programa, retornando ao sistema operacional**.

Abaixo é apresentado um programa que utiliza esta função.

```
# include <stdio.h>
void main(void)
{
        char resposta;
        printf ("Digite um texto qualquer ou <S> para sair: \n");
        for (;;)
        {
                resposta = getchar();
                if (resposta == 'S' || resposta == 's')
                        exit();
                else
                        putchar(resposta);
        }
}
```

11. ESTRUTURAS DE DADOS COMPOSTAS

11.1. VETORES E MATRIZES

Vetores nada mais são que matrizes unidimensionais. Portanto, a única diferença na declaração e manipulação entre vetores e as demais matrizes, é a quantidade de dimensões envolvidas. Estes tipos de estrutura são caracterizados por possuírem **todos os elementos de um mesmo tipo de dado**.

A sintaxe geral de sua declaração é:

tipo_dado nome_variavel [tamanho1] [tamanho2]... [tamanhoN];

sendo:

- ***N*** o número de dimensões da estrutura (no caso de vetor, ***N = 1***);
- ***tamanhoX*** o tamanho da dimensão **X** da estrutura ***nome_variavel***.

Esta declaração permite que o compilador C reserve o espaço na memória suficientemente grande para armazenar o número de células especificadas pelo tamanho de cada dimensão. Isto é, caso o compilador encontre a seguinte declaração:

int NUMEROS [10], MATRIZ [2][2];

ele irá reservar, de maneira contígua, 20 bytes na memória (2 bytes para cada inteiro vezes 10) para a variável *NUMEROS* e 8 bytes (2 x 2 x 2) para a variável *MATRIZ*.

A atribuição ou utilização do valor de uma das células de vetores ou matrizes pode ser feita através da sintaxe:

nome_variavel[numeração1] [numeração2]... [numeraçãoN] = valor;
ou
variável = nome_variavel[numeração1] [numeração2]... [numeraçãoN];

sendo:

numeraçãoX a posição da célula dentro da dimensão **X** da estrutura ***nome_variavel***.

Na linguagem C, **esta numeração começa sempre em zero**. Isto significa que, no exemplo anterior, os dados de *NUMEROS* serão indexados de 0 a 19, enquanto que MATRIZ possuirá 04 valores indexados, respectivamente, pelos pares ordenados (0,0), (0,1), (1,0) e (1,1). Entretanto, **o compilador C não verifica se o índice usado no programa está dentro dos limites válidos** (este cuidado deve ser tomado pelo programador). Quando não é dada a devida atenção aos limites de validade para os índices, corre-se o risco de ter variáveis sobrescritas ou do computador travar.

Assim como as demais variáveis, **estes tipos de estrutura também podem ser inicializados na declaração**, como segue:

```
int VETOR [5] = {5, 10, 15, 20, 25};
char TEXTO [2] [2] = {‘a’, ‘b’, ‘c’, ‘d’}; /* (0,0) = ‘a’; (0,1) = ‘b’; (1,0) = ‘c’ e (1,1) = ‘d’ */
```

Quando a estrutura é inicializada na declaração, o tamanho de uma das dimensões pode ser omitida, pois o mesmo será considerado a partir da quantidade de valores descritos entre chaves. Assim, a declaração acima poderia ser reescrita da seguinte forma:

```
int VETOR [] = {5, 10, 15, 20, 25};
char TEXTO [] [2] = {‘a’, ‘b’, ‘c’, ‘d’};
```

A seguir é apresentado um programa-exemplo que utiliza vetores.

```
#include <stdio.h>
void main (void)
{
        int num[100];  /* Declara um vetor de inteiros de 100 posições */
        int count=0;
        int totalnums;
```

```
        do
        {
                printf ("Entre com um número não nulo ou digite 0 p/ terminar: ");
                scanf ("%d",&num[count]);
                count++;
        } while ((num[count-1] != 0) && (count < 100));
        totalnums=count;
        printf ("\nOs números digitados foram:\n");
        for (count=0 ; count < totalnums ; count++)
                printf (" %d",num[count]);
}
```

Neste programa, a entrada de números é feita até que o usuário digite zero ou entre com 100 números não nulos. Os números são armazenados no vetor *num*. A cada número armazenado, a variável *count,* utilizada como índice/contador do vetor, é incrementada. Ao sair do primeiro *"loop"*, a quantidade de números digitados pelo usuário é armazenada na variável *totalnums*. Por fim, todos os números são impressos.

Abaixo é dado um exemplo do uso de matriz. Nele, a estrutura MATRIZ é preenchida, seqüencialmente, com os números de 1 a 200.

```
#include <stdio.h>
void main (void)
{
int MATRIZ [20][10];
int i,j,cont;
cont=1;
for (i=0;i<20;i++)
        for (j=0;j<10;j++)
        {
        MATRIZ[i][j]=cont;
        cont++;
      }
}
```

Vale lembrar que, tanto a leitura quanto o preenchimento de uma matriz, **o índice mais à direita varia mais rapidamente que o seu antecessor** e assim por diante até o índice mais à esquerda.

11.2. REGISTROS

Um registro é uma estrutura de dados heterogênea, ou seja, formada por uma coleção de variáveis que podem assumir tipos diferentes de dados, inclusive os tipos compostos (vetores, matrizes e registros). Neste tipo de estrutura, **as variáveis são agrupadas juntas sob um único nome** para a conveniência de manipulação.

Um exemplo tradicional é um registro de folha de pagamento, onde um empregado é descrito por um conjunto de atributos, tais como: nome, endereço, número do seguro social, salário, etc. Alguns desses atributos também podem ser uma estrutura, tal como o endereço que possui os atributos: logradouro, número, bairro, complemento, etc.

DECLARAÇÃO DE REGISTROS

Na linguagem C, um registro é declarado através da palavra reservada ***struct***. Esta palavra-chave **introduz uma lista de declarações entre chaves**. Um identificador opcional chamado "**nome da estrutura**" (*structure tag*) pode seguir a palavra ***struct***. Ele **dá nome ao registro e pode ser usado como uma abreviatura para a declaração** do registro em outras partes do programa.

As variáveis de um registro são chamadas de membros do registro. Um membro pode ter o mesmo nome de uma outra variável do programa, desde que esta não pertença ao mesmo registro. Entretanto, por uma questão de clareza, esta prática não é recomendada.

A seguir é apresentado um exemplo com a declaração do registro folha de pagamento.

```
struct FOLHA
{
        char NOME[50];

        struct END
        {
                char LOGRADOURO[50];
                int NRO;
                char BAIRRO[20];
                char COMPLEMENTO[20];
        } ENDERECO;
        long int NRO_SEG;
        float SALARIO;
};
```

Note que, no exemplo acima, dentro do registro FOLHA existe o registro END, o qual foi atribuído à variável ENDERECO. Isto ocorre porque, como qualquer outra declaração de variável, **após a chave de fechamento (}) da lista de membros do registro pode ser declarada a lista de variáveis que receberão este tipo de estrutura**.

Exemplo:

```
struct { ... } x, y, z;
```

é sintaticamente análoga a:

```
int x, y, z;
```

no sentido de que cada comando declara x, y e z como variáveis do tipo dado e faz com que seja reservado o espaço de memória necessário ao armazenamento das mesmas.

Uma declaração de registro sem a lista de variáveis só descreve o formato da estrutura, mas não aloca área de armazenamento. Se o registro tem nome (identificador), este pode ser usado depois, na definição de ocorrências do registro, como no exemplo:

```
struct FOLHA FOLHA_PAGTO;
```

Esta declaração define a variável FOLHA_PAGTO como sendo do tipo registro FOLHA. Normalmente, **a declaração do registro é feita no início do programa** (declaração global), enquanto que **a declaração das variáveis associadas a este tipo de estrutura é realizada nas respectivas funções** (declaração local ou argumentos formais).

Durante a declaração de uma variável do tipo registro, **pode-se inicializar os valores dos membros da variável**. Para isto, **basta utilizar o sinal de igual, seguido de um conjunto de constantes** (uma para cada membro, na mesma ordem da declaração do registro) **delimitado por chaves**. Um exemplo de inicialização é apresentado abaixo:

```
struct FOLHA FOLHA_PAGTO = {"José",{"Rua x", 13, "Pacaembu", "Casa B" }, 123, 900.00};
```

MANIPULAÇÃO DE MEMBROS DE UM REGISTRO

Um membro de registro é referenciado em uma expressão através da construção:

variável_registro.nome_membro

O operador ponto "." conecta o nome da variável do tipo registro e o nome do membro que será utilizado na expressão. Abaixo é apresentado alguns exemplos de uso deste operador:

```
printf ("O nome do funcionário é %s", FOLHA_PAGTO.NOME);
DESCONTO_IR = FOLHA_PAGTO.SALARIO * 0.25;
scanf ("%li", FOLHA_PAGTO.NRO_SEG);
printf ("O bairro do funcionário é %s", FOLHA_PAGTO.ENDERECO.BAIRRO);
```

Como pode ser observado no último exemplo, **a referência de membros de registros aninhados é feita através da identificação ordenada dos registros envolvidos**, do mais externo até o mais interno, todos separados por ponto; **seguida pelo nome do membro** do registro mais interno que será utilizado.

Na linguagem C, quando trabalhamos com dois registros do mesmo tipo (mesma estrutura), **podemos copiar os valores de um registro para o outro** através do comando:

registro1 = registro2;

Este comando copia o ***registro2*** para o ***registro1***, campo a campo.

MATRIZES DE REGISTRO

Um registro é como qualquer outro tipo de dado no C. Portanto, podemos criar matrizes de registros. Para isto, devemos utilizar a seguinte declaração:

struct nome_registro nome_variavel [tamanho1] [tamanho2]... [tamanhoN];

Sendo que, **a quantidade de colchetes** que serão utilizados na declaração da matriz de registro **corresponde ao número de dimensões da matriz**.

Exemplo:

```
void main(void)
{
        struct FOLHA FOLHA_PAGTO[100];
        int i;
        for (i = 0; i < 100; i++)
                FOLHA_PAGTO[i].NRO_SEG = i+1;
}
```

Neste exemplo, será alocado espaço de memória para 100 instâncias do registro *FOLHA*, associando este espaço à variável *FOLHA_PAGAMENTO*. Além disso, o *NRO_SEG* de cada instância receberá um número equivalente ao índice da instância + 1.

11.3. COMANDO TYPEDEF

O comando ***typedef*** **define um novo nome (apelido ou sinônimo) para um tipo de dado/estrutura já existente**. Sua forma geral é:

typedef tip_dado apelido;

No exemplo abaixo, a palavra *inteiro* passa a representar o tipo ***int*** e, portanto, pode ser utilizada nas declarações das variáveis deste tipo.

```
#include <stdio.h>
typedef int inteiro;
void main(void)
{
        inteiro NRO;
        scanf ("%d", &NRO);
        printf ("O número digitado foi %i",, NRO);
}
```

Este comando é muito utilizado para dar nome a tipos complexos, como registros, pois facilita a utilização desta estrutura de dados no restante do programa.

Exemplo:

```
#include <stdio.h>
typedef struct tipo_endereco  {
                char rua [50];
                int numero;
                char bairro [20];
                char cidade [30];
                char sigla_estado [3];
                long int CEP;
                } tp_endereco;
struct tipo_cadastro {
        char nome [50];
        long int telefone;
        tp_endereco endereco;
        };
typedef struct tipo_cadastro cadastro;
void main(void)
{
cadastro ALUNOS;
ALUNOS.nome = "Joaquim Manoel";
ALUNOS.endereco.cidade = "Uberlândia";
...
}
```

Observe que não é mais necessário usar a palavra chave ***struct*** na declaração das variáveis para os registros *tipo_endereco* e *tipo_cadastro*. Agora, basta usar os apelidos *tp_endereco* e *cadastro*.

12. STRINGS

12.1. MANIPULAÇÃO DE STRINGS

As strings (cadeia de caracteres) são o uso mais comum para os vetores (**string é um vetor de chars**). É importante lembrar que as strings têm o seu último elemento como um '**\0**' (terminador nulo). Portanto, o tamanho de uma string é composto pelos caracteres digitados mais o terminador nulo '**\0**'. A declaração geral para uma string é:

char nome_da_string [tamanho];

A biblioteca padrão do C possui diversas funções que manipulam strings. Estas funções são úteis, pois não se pode, por exemplo, igualar duas strings diretamente (string1 = string2;). **As strings devem ser igualadas elemento a elemento**.

Quando vamos fazer programas que tratam de string muitas vezes podemos fazer bom proveito do fato de que uma string termina com '**\0**' (isto é, o número inteiro 0). Veja, por exemplo, o programa abaixo que serve para igualar duas strings (isto é, copia os caracteres de uma string para o vetor da outra):

```
#include <stdio.h>
void main (void)
{
        int count;
        char str1[100],str2[100];
        scanf("%s",&str1);
        for (count=0;str1[count];count++)
                str2[count]=str1[count];
        str2[count]='\0';
        printf("Você digitou: %s",str2);
}
```

A condição no for acima é baseada no fato de que a string que está sendo copiada termina em '\0'. Quando o elemento encontrado em str1[count] é o '\0', o valor retornado para o teste condicional é falso (zero). Desta forma a expressão que vinha sendo verdadeira (não zero) continuamente, torna-se falsa.

12.2. FUNÇÕES BÁSICAS DE MANIPULAÇÃO DE STRINGS

Abaixo serão apresentadas algumas funções básicas para manipulação de strings. Com exceção da função ***gets()***, as demais funções estão contidas no arquivo de cabeçalho ***string.h***

GETS()

A função ***gets()*** lê uma string do teclado. Sua forma geral é:

gets (nome_da_string);

O programa abaixo demonstra o funcionamento da função ***gets()***:

```
#include <stdio.h>
int main ()
{
```

```
        char string[100];
        printf ("Digite o seu nome: ");
        gets (string);
        printf ("\n\n Ola %s",string);
        return(0);
}
```

Repare que é válido passar para a função printf() o nome da string. Como o primeiro argumento da função printf() é uma string, também é válido fazer:

printf (string); / *Este comando simplesmente imprimirá a string.*/*

STRCPY()

A função ***strcpy()*** copia a string de origem para a string de destino. Seu funcionamento é semelhante ao do programa exemplo apresentado na *Manipulação de Strings*. Sua forma geral é:

strcpy (string_destino,string_origem);

A seguir apresentamos um exemplo de uso da função strcpy():

```
#include <stdio.h>
#include <string.h>
int main ()
{
        char str1[100],str2[100],str3[100];
        printf ("Entre com uma string: ");
        gets (str1);
        strcpy (str2,str1);       /* Copia str1 em str2 */
        strcpy (str3,"Voce digitou a string "); /* Copia "Voce digitou a string" em str3 */
        printf ("\n\n%s%s",str3,str2);
        return(0);
}
```

STRCAT()

A função ***strcat()*** concatena (agrega) o conteúdo da string de origem ao final da string de destino, permanecendo a string de origem inalterada. Esta função tem a seguinte forma geral:

strcat (string_destino,string_origem);

Abaixo segue um exemplo da utilização da função ***strcat()***:

```
#include <stdio.h>
#include <string.h>
int main ()
{
        char str1[100],str2[100];
        printf ("Entre com uma string: ");
        gets (str1);
        strcpy (str2,"Voce digitou a string ");
        strcat (str2,str1);   /* str2 armazenara' Voce digitou a string + o conteúdo de str1 */
```

```
        printf ("\n\n%s",str2);
        return(0);
}
```

STRLEN()

A função ***strlen()*** retorna o comprimento da string fornecida, **desconsiderando o terminador nulo** '**\0**'. Isto quer dizer que, de fato, o comprimento do vetor da string deve ser o inteiro retornado pela função ***strlen()*** mais um. A sintaxe geral da função é:

strlen (string);

Abaixo segue um exemplo do seu uso:

```
#include <stdio.h>
#include <string.h>
int main ()
{
        int size;
        char str[100];
        printf ("Entre com uma string: ");
        gets (str);
        size=strlen (str);
        printf ("\n\nA string que voce digitou tem tamanho %d",size);
        return(0);
}
```

STRCMP()

A função ***strcmp()*** compara as duas strings passadas como argumento. Se elas forem idênticas, a função retorna o valor zero. Caso contrário, a função retorna um valor diferente de zero. Sua forma geral é:

strcmp (string1,string2);

Um exemplo da sua utilização é apresentado a seguir:

```
#include <stdio.h>
#include <string.h>
void main (void)
{
        char str1[100],str2[100];
        printf ("Entre com uma string: ");
        gets (str1);
        printf ("\n\nEntre com outra string: ");
        gets (str2);
        if (strcmp(str1,str2))
                printf ("\n\nAs duas strings são diferentes.");
        else
                printf ("\n\nAs duas strings são iguais.");
}
```

ATOI() E ATOF()

A função ***atoi()*** (abreviatura de *alphanumeric to integer*) recebe uma cadeia de caracteres que representa um número inteiro em notação decimal e a converte no número inteiro correspondente. Sua forma geral é:

var_int = atoi (string);

De forma similar, a função ***atof()*** (abreviatura de *alphanumeric to float*) recebe uma cadeia de caracteres que representa um número real em notação decimal e a converte no número real (do tipo ***double***) correspondente. Sua sintaxe é:

var_double = atof (string);

Um exemplo da sua utilização é apresentado a seguir:

```
#include <stdio.h>
#include <stdlib.h>
void main (void)
{
        char str1[2] = “12”, str2[4] = “2.14”;
        double soma;
        soma = atoi (str1) + atof (str2);
        printf ("A soma das strings é %lf.",soma);
}
```

Ambas as funções estão declaradas no arquivo de cabeçalho ***<stdlib.h>***.

12.3. VETORES DE STRINGS

Vetores de strings são matrizes bidimensionais. Isto porque **uma string é um vetor**, portanto, ao utilizar um vetor de strings estaremos fazendo, na verdade, um vetor de vetores, ou seja, uma matriz bidimensional do tipo ***char***. Podemos ver a forma geral de um vetor de strings como sendo:

char nome_da_variável [numero_de_strings][comprimento_das_strings];

Assim, para acessar um única string, basta usar o primeiro índice, como apresentado abaixo:

printf(“%s”, nome_da_variável [índice]);

Abaixo é apresentado um programa exemplo que lê 5 strings e as exibe na tela:

```
#include <stdio.h>
void main (void)
{
        char strings [5][100];
        int count;
        for (count=0;count<5;count++)
        {
                printf ("\n\nDigite a string %d: ", count);
                gets (strings[count]);
        }
        printf ("\n\n\nAs strings que voce digitou foram:\n\n");
```

```
        for (count=0;count<5;count++)
                printf ("%s\n",strings[count]);
}
```

13. FUNÇÕES

Funções são as estruturas que permitem ao usuário separar seus programas em blocos de construção. A principal característica das funções é o fato de, uma vez escritas e depuradas (testadas), elas poderem ser reutilizadas quantas vezes forem necessárias, inclusive por outros programas.

Na linguagem C, uma função tem a seguinte forma geral:

```
tipo_retorno nome_da_função (declaração_parâmetros)
{
corpo_da_função
}
```

O ***tipo_retorno*** é o tipo de variável que a função vai retornar. O default é o tipo ***int***, ou seja, uma função para a qual não se declara o tipo de retorno é considerada como retornando um inteiro.

A ***declaração_parâmetros*** é uma lista de nomes de variáveis separadas por vírgulas que recebem os valores dos argumentos quando a função é chamada. Ela possui a seguinte forma geral:

```
tipo nome1, tipo nome2, ... , tipo nomeN
```

Note que, **o tipo deve ser especificado para cada uma das *N* variáveis de entrada**. É na declaração de parâmetros que informamos ao compilador quais serão as entradas da função (assim como informamos a saída no ***tipo_retorno***). **Cada variável descrita na *declaração_parâmetros* será tratada como uma variável local da função**. Quando uma função não tiver argumentos de entrada, a lista de parâmetros será vazia. Entretanto, **os parênteses da declaração da função são obrigatórios**.

O ***corpo_da_função*** contém a declaração das variáveis locais, bem como a seqüência de comandos que serão executados, ou seja, o bloco de comandos da função.

Quando se encontra um comando ***return***, **a função é encerrada imediatamente e**, se algum dado é informado após o ***return***, **seu valor é retornado** pela função. É importante lembrar que, **o valor de retorno fornecido tem que ser compatível com o tipo de retorno declarado para a função**. Uma função pode ter mais de um comando ***return***, entretanto, somente um será executado por chamada da função. Isto se torna claro quando pensamos que a função é terminada quando o programa chega ao primeiro comando ***return***.

Abaixo estão dois exemplos de funções:

```
/* Programa-Exemplo 1 */
#include <stdio.h>
int square (int a)
{
        return (a*a);
}
void main ()
{
```

```
        int num;
        printf ("Entre com um numero: ");
        scanf ("%d",&num);
        num=square(num);
        printf ("\n\nO seu quadrado vale: %d\n",num);
}
```

```
/* Programa-Exemplo 2 */
#include <stdio.h>
int EPar (int a)
{
        if (a%2)        /* Verifica se a e divisivel por dois */
                return 0;       /* Retorna 0 se nao for divisivel */
        else
                return 1;       /* Retorna 1 se for divisivel */
}
int main ()
{
        int num;
        printf ("Entre com numero: ");
        scanf ("%d",&num);
        if (EPar(num))
                printf ("\n\nO numero e par.\n");
        else
                printf ("\n\nO numero e impar.\n");
        return 0;
}
```

No segundo exemplo, pode ser observado o uso de mais de um comando ***return*** na função.

Como as funções retornam valores, podemos aproveitá-los para fazer atribuições e comparações condicionais (como parte de uma expressão), ou, ainda, na saída de dados. Mas é importante frisar que, em uma atribuição, **as chamadas de funções só podem aparecer no lado direito dos comandos de atribuição**.

Além disso, se uma função retorna um valor que não precisa ser aproveitado pelo programa, o mesmo pode ser despresado. Por exemplo, a função ***printf()*** retorna um inteiro que quase nunca é utilizado, portanto, ele é descartado.

13.1. USO DO TIPO VOID EM FUNÇÕES

Em inglês, void quer dizer vazio e é este o sentido deste tipo de dado. **Ele permite a construção de funções que não retornam nada (procedimentos) e/ou de funções que não têm argumentos de entrada**. A seguir são apresentados alguns exemplos de protótipos de funções que não retornam nada e/ou não possuem argumentos formais:

```
void nome_da_função (declaração_parâmetros);   /* Sem retorno */
tipo_retorno nome_da_função (void);            /* Sem argumentos de entrada */
void nome_da_função (void);                    /* Sem argumentos formais e retorno */
```

Numa função sem retorno, o uso do comando ***return*** é opcional, podendo ser omitido. A seguir é apresentado um exemplo de utilização do tipo ***void*** na declaração de funções:

```
#include <stdio.h>
void Mensagem (void);
void main (void)
{
        Mensagem();
        printf ("\tDiga de novo:\n");
        Mensagem();
        return 0;
}
void Mensagem ()
{
        printf ("Ola! Eu estou vivo.\n");
}
```

Note que na declaração da função *Mensagem()*, a palavra ***void*** é omitida dos parênteses.

A função *main()* também é uma função e como tal deve ser tratada. Por default, o compilador C retorna um inteiro para a função ***main()***. Isto pode ser interessante quando se deseja que o sistema operacional receba um valor de retorno da execução do programa. Assim sendo, pode-se utilizar a seguinte convenção: **se o programa retornar zero**, significa que ele **terminou normalmente** (sem erros); **caso contrário**, ou seja, se o programa retornar um valor diferente de zero, significa que o programa teve um término anormal (**houve algum erro**).

Se não houver interesse no retorno do programa, a função ***main()*** pode ser declarada como retornando ***void***. Alguns compiladores podem reclamar deste tipo de declaração, dizendo que main sempre deve retornar um inteiro. Caso isto ocorra, basta fazer main retornar um inteiro.

13.2. CHAMADA DE FUNÇÃO

Uma vez que as funções estejam definidas, pode-se usa-las sem se preocupar como elas foram escritas. Isto é o que denominamos ***chamada de função***. A sintaxe geral da chamada de uma função é:

nome_função(lista_valores);

sendo a ***lista_valores*** uma lista com os valores que serão atribuídos a cada um dos argumentos formais declarados na função. Sempre é bom lembrar que a quantidade e o tipo dos valores declarados na ***lista_valores*** da chamada da função devem ser compatíveis com aqueles apresentados na ***declaração_parâmetros*** da declaração da função.

Em geral, os argumentos podem ser passados para as funções de duas maneiras, conforme segue:

PASSAGEM DE PARÂMETROS POR VALOR

Na ***passagem de parâmetros por valor*** (ou simplesmente ***"chamada por valor"***), os valores dos argumentos que são passados para a função são copiados nos parâmetros formais correspondentes. Assim, **quaisquer alterações feitas nestes parâmetros formais não têm efeito nas variáveis empregadas na *chamada da função***.

O programa abaixo ilustra uma passagem de parâmetros por valor

```
#include <stdio.h>
float sqr (float num);
void main ()
{
        float nro,sq;
        printf ("Entre com um numero: ");
        scanf ("%f",&nro);
        sq = sqr(nro);
        printf ("\n\nO numero original e: %f\n",nro);
        printf ("O seu quadrado vale: %f\n",sq);
}
float sqr (float num)
{
        num = num*num;
        return num;
}
```

No exemplo acima, o parâmetro formal ***num*** da função ***sqr()*** sofre alterações dentro da função, mas a variável ***nro*** da função ***main()*** permanece inalterada.

PASSAGEM DE PARÂMETROS POR REFERÊNCIA

Outro tipo de passagem de parâmetros para uma função ocorre quando alterações nos parâmetros formais, dentro da função, alteram os valores dos parâmetros que foram passados para a função. Este tipo de chamada de função tem o nome de ***"chamada por referência"***. Este nome vem do fato de que, neste tipo de chamada, não se **passa para a função** os valores das variáveis, mas sim **suas referências** (**endereço da variável na memória**). A função usa estas referências para **alterar os valores das variáveis empregadas na chamada da função**.

Neste tipo de chamada, os parâmetros formais de uma função devem ser declarados como sendo ponteiros (utilizando o operador *****). Os ponteiros são a "*referência*" que precisamos para poder alterar a variável fora da função. Além disso, para que estes ponteiros recebam o endereço de memória das variáveis utilizadas na ***chamada da função***, é necessário colocar o operador **&** na frente de cada uma das variáveis, como apresentada no exemplo:

```
#include <stdio.h>
void sqr (float *num);
void main ()
{
        float nro;
        printf ("Entre com um numero: ");
        scanf ("%f",&nro);
        printf ("\n\nO numero original e: %f\n",nro);
        sqr(&nro);
        printf ("O seu quadrado vale: %f\n",nro);
}
void sqr (float *num)
        *num = (*num)*(*num);
```

Neste exemplo, reescrevemos o programa anterior. Nessa nova versão, é passado o endereço de memória da variável ***nro*** para a função ***sqr()***. Este endereço é copiado no ponteiro ***num***. Através

do operador ***** acessamos o conteúdo apontado por este ponteiro e realizamos a operação desejada, alterando-o. Neste caso, este conteúdo nada mais é que o valor armazenado em ***nro***, que, portanto, está sendo modificado.

Note que, uma vez que o valor da função foi armazenado no endereço de memória de ***nro***, não é necessário nenhum retorno.

Este tipo de chamada é a mesma utilizada pela função ***scanf()***. Isto porque, esta função precisa alterar o valor da variável passada como parâmetro, atribuindo o novo valor digitado pelo usuário.

13.3. RECURSIVIDADE

As funções em C podem chamar umas às outras para executar uma tarefa específica. Um caso especial de chamada de funções ocorre quando uma função chama a si mesma. Este processo é denominado de **"recursão"** e a função é chamada ***função recursiva***.

Na concepção de uma função recursiva, **a primeira providência é definir um critério de parada para a recursão**, ou seja, determinar quando a função deverá parar de chamar a si mesma. Isto impede que a recursão se torne infinita e, consequentemente, o programa não tenha fim.

Um bom exemplo de uma função recursiva é a função que calcule o fatorial de um número inteiro:

```
#include <stdio.h>
unsigned int fat(unsigned int n)
{
if (n > 1 )
return n*fat(n-1);
else
return 1;
}
void main(void)
{
        unsigned int NRO;
        printf("\n\nDigite um valor inteiro positivo: ");
        scanf("%u", &NRO);
        printf("\nO fatorial de %d e' %d", NRO, fat(NRO));
}
```

Neste exemplo, enquanto ***n > 1***, a função ***fat()*** chama a si mesma, cada vez com um valor menor. O critério de parada da função é ***n ≤ 1***.

Uma função recursiva quase sempre consome mais memória e tem um processamento mais demorado que sua versão não recursiva. Isto ocorre porque, a memória consumida na chamada de uma função só é liberada na conclusão da mesma, portanto, o computador aloca muito mais memória quando a função é recursiva. Entretanto, **o código recursivo é mais compacto e, às vezes, mais fácil de entender**. Além disso, há certos algoritmos que são mais eficientes quando feitos de maneira recursiva. Como exemplo, a construção de alguns tipos de estruturas de dados abstratos, tais como árvores que até mesmo a sua definição é recursiva.

13.4. MATRIZES COMO ARGUMENTO DE FUNÇÕES

Quando os argumentos de uma função são matrizes, tem-se uma exceção à convenção de passagem de parâmetros com chamada por valor. Neste caso, apenas o endereço da matriz é

passado, não uma cópia da matriz inteira, ou seja, quando se chama uma função com argumento matriz, um ponteiro para o primeiro elemento da matriz é passado para a função.

Existem três maneiras de se declarar um parâmetro que irá receber um ponteiro para matriz:

1. Declarando-o como uma matriz;

Tipo_retorno nome_função (tipo_matriz nome_matriz [dim1] [dim2]... [dimN]);

2. Especificando uma matriz sem dimensão;

Tipo_retorno nome_função (tipo_matriz nome_matriz [] []... []);

3. Declarando-o como um ponteiro, o que é o mais comum.

Tipo_retorno nome_função (tipo_matriz *nome_matriz [dim2]... [dimN]);

Como exemplo, seja o vetor de inteiros ***matrx [50]*** o argumento de uma função ***func()***. Podemos declarar ***func()*** das seguintes maneiras:

```
void func (int matrx[50]);
void func (int matrx[]);
void func (int *matrx);
```

Nos três exemplos, existirá um ponteiro (***int ****) chamado ***matrx*** dentro de ***func()***. Ao passarmos uma matriz para uma função, na realidade estamos passando um ponteiro. Neste ponteiro é armazenado o endereço do primeiro elemento da 1ª dimensão. Isto significa que não é feita uma cópia, elemento a elemento da matriz. Assim, é possível alterar o valor dos elementos da matriz dentro da função.

13.5. ARGUMENTOS PARA A FUNÇÃO PRINCIPAL *MAIN()*

Algumas vezes é desejável passar informações para um programa no início de sua execução. Nestes casos, estas informações devem ser passadas para a função ***main()*** via argumentos da linha de comando e recebidas pela função através de seus parâmetros formais. A função ***main()*** **pode ter dois parâmetros formais já pré-determinados**. Os parâmetros ***argc*** e ***argv*** dão ao programador **acesso à linha de comando do sistema operacional pela qual o programa foi chamado**. A declaração de uma função ***main()*** com estes parâmetros é:

```
tipo_retorno main (int argc,char *argv[]);
```

O ***argc*** (*argument count*) é um inteiro e **possui o número de palavras digitadas na linha de comando**. Seu valor é sempre, pelo menos, igual a 1, pois **o nome do programa é contado como sendo o primeiro argumento**. Ele tem por finalidade **indicar quantos elementos temos em *argv***.

O ***argv*** (*argument values*) é um ponteiro para uma matriz de strings. **Cada string desta matriz é um dos parâmetros da linha de comando**, separados por espaços em branco. O ***argv[0]*** **sempre aponta para o nome do programa** (que, como já foi dito, é considerado o primeiro argumento).

No exemplo abaixo, foi escrito um programa chamado *data* que faz uso dos parâmentros ***argv*** e argc. Este programa deverá receber da linha de comando os inteiros correspondentes ao dia, mês e ano, no formato abreviado, e imprimi-la por extenso.

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

```
void main(int argc, char *argv[])
{
        int mês;
        char *nm_mes[] = {"Janeiro", "Fevereiro", "Março", "Abril", "Maio", "Junho", "Julho",
         "Agosto", "Setembro", "Outubro",  "Novembro", "Dezembro"};

        /* O if abaixo testa se o numero de parametros fornecidos está correto:
                1º - nome do programa; 2º - dia; 3º - mês; e 4º - ano */
        if(argc == 4)
        {
                /* argv contém strings. A string referente ao mês deve ser transformada em
                um inteiro. Para isto, utiliza-se a função atoi() */
                mes = atoi (argv[2]);

                if (mes < 1 || mes > 12)  /* Testa se o mês é válido */
                        printf("O mês deveria estar dentro do intervalo 1 e 12.");
                else
                        printf("\n%s de %s de 19%s", argv[1], nomemes[mes-1], argv[3]);
        }
        else
                printf("Número de parâmetros inválido.");
}
```

Neste exemplo, ao digitar a linha de comando:

```
data 19 04 1999
```

o programa apresentará como saída:

```
19 de abril de 1999
```

13.6. CONSIDERAÇÕES FINAIS

É recomendável **implementar funções da maneira mais geral possível**, pois isto facilita a sua reutilização e entendimento. Além disso, sempre que possível, **deve-se evitar o uso variáveis globais nas funções**, pois isto dificulta o reaproveitamento da função por outros programas, bem como pode causar efeitos colaterais decorrentes da alteração indevida no valor destas variáveis em outras partes do programa.

Por fim, quando um programa deve apresentar um bom desempenho (*performance*) de processamento, seria bom implementá-lo sem nenhuma (ou com o mínimo de) chamadas a funções, porque uma chamada a uma função consome tempo e memória.

14. ARQUIVOS

O sistema de manipulação de arquivos do ANSI C é composto por uma série de funções interelacionadas, cujos protótipos estão reunidos em ***stdio.h***. Todas estas funções trabalham com o conceito de "**ponteiro de arquivo**". **Um ponteiro de arquivo aponta para uma estrutura que contém informações sobre o arquivo**, como o local de um *buffer*, a posição do caracter corrente no *buffer*, se o arquivo está sendo lido ou gravado e se foram encontrados erros ou o fim do arquivo. Entratanto, o programador não presisa saber os detalhes desta estrutura, pois a sua

declaração já está definida no arquivo de cabeçalho ***stdio.h*** e é chamada ***FILE***. Portanto, para se cirar um ponteiro de arquivo, basta declará-lo através da sintaxe:

FILE *fp;

Onde:

- ***FILE*** é o tipo de dado especial, definido como um ***typedef*** dentro do arquivo ***stdio.h***;
- ***fp*** é o nome do ponteiro para arquivo.

14.1. ARQUIVOS PRÉ-DEFINIDOS

Na linguagem C, um arquivo é um conjunto de dados que pode ser armazenado e/ou obtido desde um arquivo em disco até um periférico (ex: teclado ou impressora).

Quando se começa a execução de um programa, o sistema automaticamente abre alguns arquivos pré-definidos:

- **stdin:** dispositivo de entrada padrão (geralmente o teclado)
- **stdout:** dispositivo de saída padrão (geralmente o vídeo)
- **stderr:** dispositivo de saída de erro padrão (geralmente o vídeo)
- **stdaux:** dispositivo de saída auxiliar (em muitos sistemas, associado à porta serial)
- **stdprn :** dispositivo de impressão padrão (em muitos sistemas, associado à porta paralela)

Cada uma destas constantes pode ser utilizada como um ponteiro para ***FILE***, para acessar os respectivos periféricos associados. Por exemplo:

```
ch =getc(stdin);        /* efetua a leitura de um caracter a partir do teclado */
putc(ch, stdprn);       /* imprime o caracter lido na impressora */
```

14.2. ABERTURA DE ARQUIVOS

A abertura ou criação de um arquivo no C é feita através da função ***fopen()***. Ela retorna o ponteiro de arquivo associado ao arquivo. O formato geral da função é:

fp = fopen(“nome_arquivo”, “modo_abertura”);

Onde:

- ***fp*** é o nome do ponteiro para arquivo (ponteiro do tipo **FILE**);
- ***nome_arquivo*** é uma ***string*** que contém o nome do arquivo que será manipulado. Este nome deverá ser válido no sistema operacional que estiver sendo utilizado (ex: no DOS os nomes devem ter no máximo 12 caracteres, incluindo o ponto e a extensão do arquivo);
- ***modo_abertura*** é uma ***string*** que determina como o arquivo será aberto e, portanto, utilizado. A tabela abaixo mostra os possíveis valores para esta ***string***.

VALORES VÁLIDOS PARA O MODO DE ABERTURA DE ARQUIVOS

Modo	Significado
"r"	Abre um arquivo texto para leitura. O arquivo deve existir antes de ser aberto.
"w"	Abrir um arquivo texto para gravação. Se o arquivo não existir, ele será criado. Se já existir, o conteúdo anterior será destruído.
"a"	Abrir um arquivo texto para gravação. Se ele já existir, os dados serão adicionados no fim do arquivo ("append"). Caso contrário, um novo arquivo será criado.
"r+"	Abre um arquivo texto para leitura e gravação. O arquivo deve existir e pode ser modificado.
"w+"	Cria um arquivo texto para leitura e gravação. Se o arquivo existir, o conteúdo anterior será destruído. Se não existir, será criado.
"a+"	Abre um arquivo texto para gravação e leitura. Se o arquivo já existir, os dados serão adicionados no fim do mesmo. Caso contrário, um novo arquivo será criado.
"rb"	Abre um arquivo binário para leitura (similar ao modo "r").
"wb"	Cria um arquivo binário para escrita (similar ao modo "w").
"ab"	Acrescenta dados ao final do arquivo binário (similar ao modo "a").
"r+b"	Abre um arquivo binário para leitura e escrita (similar ao modo "r+").
"w+b"	Cria um arquivo binário para leitura e escrita (similar ao modo "w+")
"a+b"	Acrescenta dados ou cria uma arquivo binário para leitura e escrita (similar ao modo "a+").

No exemplo abaixo, é aberto um arquivo binário para escrita:

```
FILE *fp;                       /* Declaração do ponteiro de arquivo */
fp=fopen ("exemplo.bin","wb");  /* abertura do arquivo se chama exemplo.bin e está
                                   localizado no diretório corrente */
if (!fp)
    printf ("Erro na abertura do arquivo.");
```

A condição **!fp** testa se o arquivo foi aberto com sucesso porque, no caso de um erro, a função ***fopen()*** retorna um ponteiro nulo (**NULL**).

*OBS: A macro **NULL** é definida em **stdio.h** como '**\0**'.*

O padrão ANSI determina que pelo menos 8 arquivos podem ser abertos simultaneamente. Entretanto, a maioria dos compiladores permitem muito mais que isso.

14.3. FECHAMENTO DE ARQUIVOS

Quando acabamos de usar um arquivo que abrimos, devemos fechá-lo. Para tanto usa-se a função ***fclose()***. Esta função fecha um arquivo que tenha sido aberto através da função ***fopen()***. Sua sintaxe geral é:

```
fclose (fp);
```

Onde ***fp*** é o nome do ponteiro para o arquivo (ponteiro do tipo **FILE**) aberto pela função ***fopen()***.

O não fechamento de um arquivo pode causar vários tipos de problemas, incluindo perda de dados, destruição de arquivos e possíveis erros intermitentes no programa. Um fechamento de arquivo com ***fclose()*** também libera o bloco de controle de arquivo, deixando-o disponível para reutilização.

Fechar um arquivo faz com que qualquer caracter que tenha permanecido no "***buffer***" associado ao fluxo de saída seja gravado.

Mas, o que é este "***buffer***"?

Quando se envia caracteres para serem gravados em um arquivo, **estes caracteres são armazenados temporariamente em uma área de memória (o "*buffer*")** em vez de serem escritos em disco imediatamente. **Quando o "*buffer*" estiver cheio, seu conteúdo é escrito no disco de uma vez**. A razão para se fazer isto tem a ver com a eficiência nas leituras e gravações de arquivos. Se, para cada caracter que fossemos gravar, tivéssemos que posicionar a cabeça de gravação em um ponto específico do disco, as gravações seriam muito lentas. Com o "*buffer*", **as gravações só serão efetuadas quando houver um volume razoável de informações a serem gravadas ou quando o arquivo for fechado**.

A função retorna um **valor inteiro igual a zero quando a operação foi bem sucedida**. **Qualquer valor diferente indica erro**. Geralmente, os erros retornados estão associados à não localização da unidade de disco ou à inexistência de espaço no disco.

Outra forma de fecharmos arquivos é através da função ***exit()***. Esta função fecha todos os arquivos abertos do programa.

14.4. LEITURA E ESCRITA EM ARQUIVOS

Após a abertura de um arquivo, é possível ler ou escrever nele utilizando as funções que serão apresentadas a seguir.

ESCREVENDO CARACTERES EM ARQUIVOS

Existem duas formas equivalentes para escrever caracteres em arquivos: a macro ***putc()*** e a função ***fputc()***. Estas duas formas são equivalentes e possuem a mesma sintaxe:

```
putc (ch, fp);
fputc (ch, fp);
```

Onde:

- ***fp*** é o nome do ponteiro para o arquivo (ponteiro do tipo **FILE**) aberto pela função ***fopen()***;
- ***ch*** é o caracter a ser escrito no arquivo na posição apontada pelo ponteiro ***fp***.

Resumidamente, a função grava o caracter ***ch*** no arquivo apontado pelo ponteiro ***fp***, avança o ponteiro de posição no arquivo e retorna o caracter escrito. Se a função definir um ponteiro de erro, será retornado "**EOF**".

O programa a seguir lê uma string do teclado e a escreve, caractere por caractere, em um arquivo em disco (*string.txt*).

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
```

```
        FILE *fp;
        char string[100];
        int i;
        fp = fopen("string.txt","w");   /* Arquivo ASCII, para escrita */
        if(!fp)
        {
                printf( "Erro na abertura do arquivo");
                exit(0);
        }
        printf("Entre com a string a ser gravada no arquivo:");
        gets(string);
        for(i=0; string[i]; i++)
                putc(string[i], fp);       /* Grava a string, caractere a caractere */
        fclose(fp);
}
```

Após a execução deste programa, o arquivo *string.txt* é criado no diretório corrente e seu o conteúdo pode ser acessado através de qualquer editor de textos.

LENDO CARACTERES DE ARQUIVOS

Assim como no caso anterior, também existem duas formas equivalentes para ler caracteres de um arquivo: a macro ***getc()*** e a função ***fgetc()***. As duas formas são equivalentes e possuem a mesma sintaxe:

```
var = getc (fp);
var = fgetc (fp);
```

Onde:

- ***fp*** é o nome do ponteiro para o arquivo (ponteiro do tipo **FILE**) aberto pela função ***fopen()***;
- ***var*** é a variável (do tipo ***int*** ou ***char***) que receberá o caracter lido.

Em caso de fim de arquivo ou erro, será retornado "**EOF**".

O programa abaixo, ilustrando a aplicação da macro ***getc()***, lê qualquer arquivo ASCII, informado como argumento na chamada do programa, e mostra o seu conteúdo na tela:

```
/* Programa: imp_txt.c */
# include <stdio.h>
# include <stdlib.h>
void main (int argc, char *argv[])
{
        FILE *fp;
        char ch;
        if (argc != 2)
        {
                puts ("Sintaxe: imp_txt  nome_arq_ASCII");
                exit(1);
        }
```

```
        if ((fp = fopen(argv[1], "r")) == NULL)
        {
                puts ("O arquivo não pode ser aberto");
                exit(1);
        }
        ch = getc(fp) /* Lê um caracter do arquivo apontado por fp */
        printf ("\n\n Saída do arquivo %s : \n", argv[1]);
        while (ch != EOF)
        {
                putchar(ch); /* Mostra o caracter na tela */
                ch = getc(fp);
        }
        fclose(fp);
        getchar();
}
```

LENDO STRINGS DE ARQUIVOS

Para se ler uma string num arquivo podemos usar a função ***fgets()***. Sua sintaxe é:

fgets (str, tam, fp);

Onde:

- ***str*** é a variável que receberá a string lida;
- ***tam*** é o tamanho máximo da string que será lida;
- ***fp*** é o nome do ponteiro para o arquivo aberto, de onde a string será lida.

Esta função lê a string até que um caracter de nova linha seja lido ou ***tamanho-1*** caracteres tenham sido lidos. Se o caracter de nova linha ('**\n**') for lido, ele fará parte da string, o que não acontecia com a função ***gets()***. A string resultante sempre terminará com '**\0**' (por isto somente ***tamanho-1*** caracteres, no máximo, serão lidos).

A função ***fgets()*** é semelhante à função ***gets()***, porém, além dela poder fazer a leitura a partir de um arquivo de dados e incluir o caracter de nova linha na string, ela ainda **especifica o tamanho máximo da string de entrada**. Como já visto, a função ***gets()*** não tinha este controle, o que poderia acarretar erros de "***estouro de buffer***". Portanto, levando em conta que o ponteiro **fp** pode ser substituído por ***stdin***, como vimos acima, uma alternativa ao uso de ***gets()*** é usar a seguinte construção:

fgets (str, tam, stdin);

Sendo:

- ***str*** a string que se está lendo;
- ***tam*** o número máximo de caracteres que podem ser lidos (*tam = tamanho de str - 1*). Esta subtração se dá por causa do caracter '**\0**'.

ESCREVENDO STRINGS EM ARQUIVOS

Para se escrever uma string num arquivo podemos usar a função ***fputs()***. Sua sintaxe é:

fputs (str, fp);

A função ***fputs()*** acessa os caracteres da string ***str*** (exceto o caracter especial terminador nulo – "**\0**") e grava-os no arquivo de saída apontado pelo ponteiro ***fp***. Ela retorna um valor negativo se não tiver um ponteiro de erro definido, caso contrário, retorna **EOF**.

No programa-exemplo abaixo, uma string é lida do teclado e escrita no arquivo demonimando *TESTE*. O final do programa ocorrerá quando o usuário inserir uma linha em branco.

```
# include <stdio.h>
# include <stdlib.h>
void main (void)
{
        char str[80];
        FILE *fp;
        fp = fopen("TESTE", "w");
        if (fp == NULL)
        {
                printf ("O arquivo não pode ser aberto \n");
                exit(1);
        }
        do {
                printf ("Entre com uma string ou tecle <ENTER> para terminar: \n");
                fgets (str, 79, stdin);
                fputs (str, fp);
        } while (*str != '\n');
        fclose (fp);
}
```

LENDO E ESCREVENDO BLOCOS DE DADOS DE ARQUIVOS

Para **ler e escrever tipos de dados maiores que um byte**, o sistema de arquivo ANSI C fornece as funções ***fread()*** e ***fwrite()***. Essas funções permitem a leitura e a escrita de blocos de qualquer tipo de dado.

Função *fread()*

A sintaxe de chamada à função ***fread()*** é:

fread (buffer, num_bytes, cont, fp);

Onde:

- ***buffer*** é um ponteiro para uma região de memória que receberá os dados lidos do arquivo;
- ***num_bytes*** é o tamanho (em bytes) de cada dado que será lido do arquivo;
- ***cont*** determina a quantidade de elementos (cada um de comprimento ***num_bytes***) que serão lidos do arquivo;
- ***fp*** é o ponteiro para um arquivo aberto pelo comando ***fopen()***.

Assim, a quantidade total de bytes normalmente lidos do arquivo é dada pela expressão:

num_bytes * cont

A função ***fread()*** devolve o número de elementos efetivamente lidos. Este valor pode ser menor que ***cont*** se o final de arquivo for atingido ou ocorrer algum erro durante a leitura.

Função *fwrite()*

A função ***fwrite()*** possui os mesmos argumentos que a função ***fread()***, contudo, neste caso, os argumentos possuem significados diferentes. A sintaxe da função é:

fwrite (buffer, num_bytes, cont, fp);

Onde:

- ***buffer*** é um ponteiro para os dados que serão escritos no arquivo;
- ***num_bytes*** é o tamanho (em bytes) de cada dado que será escrito no arquivo;
- ***cont*** determina a quantidade de elementos (cada um de comprimento ***num_bytes***) que serão escritos no arquivo;
- ***fp*** é o ponteiro para um arquivo aberto pelo comando ***fopen()***.

A função ***fwrite()*** devolve o número de elementos efetivamente escritos. Este valor será igual a ***cont***, a menos que ocorra algum erro durante a processo.

Quando o arquivo for aberto para dados binários, as funções ***fread()*** e ***fwrite()*** podem ler e escrever qualquer tipo de dado.

O programa-exemplo abaixo escreve e, em seguida, lê um ***double***, um ***int*** e um ***long int*** em um arquivo do disco. Note que, neste exemplo, foi utilizado o operador ***sizeof()***. Ele retorna o tamanho em bytes de uma variável ou tipo de dado.

```
# include <stdio.h>
# include <stdlib.h>
void main (void)
{
        FILE *fp;
        double d = 12.23;
        int i = 101;
        long int l = 1230232;
        if ((fp = fopen("TEST", "wb")) = = NULL)
        {
                printf ("O arquivo não pode ser aberto \n");
                exit(1);
        }
        if ((fwrite (&d, sizeof(double), 1, fp)) != 1)
                puts("Houve um erro na escrita do arquivo – variável double");
        if ((fwrite (&i, sizeof(int), 1, fp)) != 1)
                puts("Houve um erro na escrita do arquivo – variável int");
        if ((fwrite (&l, sizeof(long int), 1, fp)) != 1)
                puts("Houve um erro na escrita do arquivo – variável long int");
        rewind(fp); /* Volta para o início do arquivo */
```

```
        if ((fread (&d, sizeof(double), 1, fp)) != 1)
                puts("Houve um erro na leitura do arquivo – variável double");
        if ((fread (&i, sizeof(int), 1, fp)) != 1)
                puts("Houve um erro na leitura do arquivo – variável int");
        if ((fread (&l, sizeof(long int), 1, fp)) != 1)
                puts("Houve um erro na leitura do arquivo – variável long int");
        printf ("%.3f %d %ld \n", d, i, l);
        fclose(fp);
}
```

Como pode ser observado, o ***buffer*** pode ser, e geralmente é, simplesmente a memória usada para armazernar uma variável (por isso a necessidade de utilizar o operador **&**).

O uso das funções ***fread()*** e ***fwrite()*** é muito útil para os programas que envolvem a leitura e escrita de tipos de dados definidos pelo usuário, especialmente registros. Por exemplo, para a variável ***folha*** definida como:

```
struct f { char nome[80];
        float salário; } folha;
```

Podemos escrever seu conteúdo em um arquivo apontado por ***fp*** através da seguinte sentença:

fwrite (&folha, sizeof(struct f), 1, fp);

FLUXOS PADRÃO

Os fluxos padrão em arquivos permitem ao programador ler e escrever em arquivos da maneira similar a qual ele lê e escreve na tela. Para isto, são utilizadas as funções ***fscanf()*** e ***fprintf()***, respectivamente.

Função *fprintf()*

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

fprintf (fp, "string_controle", lista de argumentos);

Note que, a única diferença na sintaxe de ***fprintf()*** em relação a ***printf()*** é a especificação do ponteiro para o arquivo de destino (***fp***).

Função *fscanf()*

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

fscanf (fp, "string de controle", lista de endereços de memória das variáveis);

Novamente, a única diferença da sintaxe desta função para a da função ***scanf()*** é a especificação do ponteiro para o arquivo de destino (***fp***).

Abaixo é apresentado um exemplo de utilização destas funções. Neste programa, são lidos uma string e um número inteiro do teclado, os quais são gravados no arquivo. Em seguida, se lê a string e o número inteiro do arquivo e os apresenta na tela.

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

```
void main(void)
{
        FILE *fp;
        char str[80];
        int t;
        if ( !( fp = fopen(“TESTE”,"w") ) )      /* Abre o arquivo para gravação */
        {
                printf("Arquivo não pode ser aberto. \n");
                exit(1);
        }
        printf (“Entre com uma string e um numero: \n”);
        fscanf (stdin, “%s  %d”, str, &t);       /* Lê do teclado */
        fprintf (fp," %s  %d", str, t);          /* Escreve no arquivo */
        fclose (fp);      /* Fecha o arquivo */
        if ( !( fp = fopen(“TESTE”,"r") ) )      /* Abre o arquivo para leitura */
        {
                printf("Arquivo não pode ser aberto. \n");
                exit(1);
        }
        fscanf (fp,"%s  %d", str, &t);           /* Lê do arquivo */
        fprintf (stdout, "%s  %d", str, t);      /* Escreve na tela */
        fclose (fp);      /* Fecha o arquivo */
}
```

14.5. REMOÇÃO DE ARQUIVOS

Na linguagem C, pode-se apagar arquivos usando a função ***remove()***. Sua sintaxe é:

remove (nm_arquivo);

Sendo:

- ***nm_arquivo:*** o nome do arquivo a ser removido.

Esta função retorna zero caso a remoção seja bem sucedida e um valor diferente de zero, caso contrário.

Como esta operação não pode ser desfeita, é recomendado confirmar, junto ao usuário, a sua execução antes da efetiva remoção. Abaixo, é apresentado um programa que apaga um arquivo especificado na linha de comando.

```
# include <stdio.h>
# include <stdlib.h>
# include <ctype.h>
void main (int argc, char *argv[])
{
        char CONFIRMA;
        if (argc != 2)
        {
                printf (“O nome do arquivo não foi informado. \n”);
```

```
                exit(1);
        }
        printf ("Confirma a remoção do arquivo %s (S/N) \?", argv[1]);
        scanf("%c", &CONFIRMA);
        if ( toupper(CONFIRMA) = = 'S' )
                if ( remove(argv[1]) != 0)
                {
                        puts ("O arquivo não pode ser apagado.");
                        exit(1);
                }
}
```

14.6. LIMPEZA DO BUFFER

A função ***fflush()*** é usada para esvaziar o conteúdo de um ***buffer*** de saída. Sua sintaxe é:

fflush(fp);

Sendo:

- ***fp:*** um ponteiro de arquivo válido.

Esta função escreve o conteúdo de qualquer dado existente no ***buffer*** do arquivo associado a ***fp***. Se não for informado o ponteiro ***fp***, a função descarrega todos os arquivos abertos para saída.

Quando a operação é realizada com sucesso, a função retorna zero. Caso contrário, retorna **EOF**.

O exemplo abaixo ilustra o uso desta função:

```
# include <stdio.h>
# include <string.h>
void main (void)
{
        FILE *stream;
        static char msg[] = "Isto é um teste ";
        if ( (stream = fopen ("ARQ.DAT", "w") ) == NULL)
        {
                printf ("O arquivo ARQ.DAT não pode ser aberto. \n");
                exit(1);
        }
        fwrite (msg, strlen(msg), 1, stream);
        puts ("Pressione qualquer tecla para descarregar o arquivo ARQ.DAT");
        getchar();
        if ( fflush (stream) == 0 )
                puts ("O arquivo foi descarregado com sucesso.");
        else
                puts ("O arquivo não pode ser descarregado.");
        fclose (stream);
}
```

14.7. FUNÇÕES DE REPOSICIONAMENTO

Toda vez que estamos trabalhando com arquivos, há uma espécie de posição atual no arquivo. Esta é a posição de onde será lido ou escrito o próximo caractere. Normalmente, **num acesso seqüêncial**, não precisamos mexer nesta posição, pois quando lemos um dado, **a posição no arquivo é automaticamente incrementada**. Entretanto, **num acesso randômico**, muitas vezes **é necessário reposicionar o indicador de posição do arquivo** para a obtenção do dado desejado.

FUNÇÃO *REWIND()*

A função ***rewind()*** reposiciona o indicador de posição do arquivo (ponteiro) para o início do arquivo especificado pelo ponteiro passado como argumento. Sua sintaxe é:

rewind(fp);

O uso desta função provoca o mesmo efeito do fechamento e abertura do arquivo. Entretanto, o uso desta função é mais eficiente quanto à performance.

FUNÇÃO *FSEEK()*

Para se fazer procuras e acessos randômicos em arquivos usa-se a função ***fseek()***. Ela modifica o ponteiro de posição do arquivo. Sua sintaxe é:

fseek (fp, num_bytes, origem);

Onde:

- ***fp*** é o ponteiro de arquivo retornado por uma chamada à ***fopen()***;
- ***Num_bytes*** é um inteiro longo que determina o número de bytes a partir de ***origem*** que será a nova posição do ponteiro;
- ***origem*** é uma das macros descritas na tabela abaixo, e que estão definidas no arquivo de cabeçalho ***stdlib.h***.

NOMES DAS MACROS UTILIZADAS NA FUNÇÃO FSEEK()

Nome	Valor	Significado
SEEK_SET	**0**	Início do arquivo
SEEK_CUR	**1**	Ponto atual no arquivo
SEEK_END	**2**	Fim do arquivo

Abaixo é apresentado um exemplo de utilização desta função:

```
# include <stdio.h>
# include <stdlib.h>
void main (int argc, char *argv[])
{
        FILE *fp;
        if (argc != 3)
        {
                printf ("O nome do arquivo e/ou o número de bytes não informados. \n");
                exit(1);
```

```
        }
        if ( ( fp = fopen(argv[1], "r") ) == NULL)
        {
                printf ("Arquivo %s não pode ser aberto \n", argv[1]);
                exit(1);
        }
        if ( fseek (fp, atol(argv[2]), SEEK_SET) )
        {
                printf ("Erro na busca. \n");
                exit(1);
        }
        printf ("O %d.o caracter é %c \n", (atol(argv[2]) + 1), getc(fp));
        fclose(fp);
}
```

14.8. OUTRAS FUNÇÕES

MACRO *FEOF()*

EOF ("*End Of File*") **indica o fim de um arquivo**. Assim, é possível descobrir se o programa chegou ao final do arquivo através da comparação do caracter lido com o identificador **EOF**, descrito no arquivo ***stdio.h***. Esta comparação é ilustrada no exemplo abaixo:

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

Outra forma de descobrir se o fim de arquivo foi atingido é através do uso da macro ***feof()***. Ela retorna um valor diferente de zero quando for encontrado o final do arquivo associado ao ponteiro de arquivo especificado. Caso contrário, retornará zero. Sua sintaxe é:

```
feof (fp);
```

Deste modo , podemos reescrever o programa anterior:

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

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

FUNÇÃO *CLEARERR()*

A função ***clearerr()*** limpa o erro ou sinal de fim de arquivo associado ao ponteiro de arquivo. Esta função possui a seguinte sintaxe:

clearerr(fp);

Sendo:

- ***fp*** um ponteiro de arquivo associado ao arquivo onde fora encontrado **EOF** ou tenha ocorrido algum erro.

MACRO *FERROR()*

A macro ***ferror()*** **determina se uma operação com arquivo produziu um erro**. Ela retorna zero se nenhum erro ocorreu ou um número diferente de zero se houve algum erro durante o acesso ao arquivo. Sua sintaxe geral é:

ferror(fp)

Esta macro é muito útil quando desejamos verificar se o acesso a um arquivo teve sucesso, garantindo, assim, a integridade dos nossos dados. Na maioria dos casos, se um arquivo pode ser aberto, ele pode ser lido ou gravado. Porém, existem situações em que isto não ocorre. Por exemplo, pode acabar o espaço em disco enquanto gravamos, ou o disco pode estar com problemas e não conseguimos ler, etc.

No programa-exemplo a seguir, fazemos uso das macros ***ferror()***, ***feof()*** e da função ***clearerr()***. Este programa lê strings do arquivo **TESTE** e as grava no arquivo **TESTE1**.

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
        FILE *input, *output;
```

```
        char str [128];
        if ( (input = fopen ("TESTE","r")) ==NULL )
        {
                printf("\nO arquivo TESTE não pode ser aberto! ");
                exit(1);
        }
        else
        {
                if ( (output = fopen ("TESTE1","w")) ==NULL )
                {
                        printf("\nO arquivo TESTE não pode ser aberto! ");
                        exit(1);
                }
                else
                {
                        fgets (str, sizeof(str), input);
                        while ( !feof (input) )
                        {
                                if ( ferror(input) )
                                {
                                        puts (“Erro na leitura do arquivo \a”);
                                        clearerr (input);
                                }

                                if ( ferror(output) )
                                {
                                        puts (“Erro na gravação do arquivo \a”);
                                        clearerr (output);
                                }

                                fgets (str, sizeof(str), input);

                        }
                        fclose(input);
                        fclose(output);
                }
        }
}
```

FUNÇÃO *PERROR()*

A função ***perror()*** (“*Print Error*”) exibe uma mensagem de erro correspondente ao número inteiro equivalente ao código de erro identificado (***errno***). A mensagem será exibida na saída padrão de erro definida em ***“stderr”***.

A sua sintaxe é:

perror(str);

Onde:

- ***str*** é uma string que indica em que parte do programa ocorreu o erro.

No exemplo abaixo,

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
        FILE *fp;
        char string[100];
        if((fp = fopen("arquivo.txt","w")) ==NULL)
        {
                printf("\nNao consigo abrir o arquivo ! ");
                exit(1);
        }
        do
        {
                printf("\nDigite uma nova string. Para terminar, digite <enter>: ");
                gets(string);
                fputs(string, fp);
                putc('\n', fp);
                if(ferror(fp))
                {
                        perror("Erro na gravacao");
                        fclose(fp);
                        exit(1);
                }
        } while (strlen(string) > 0);
        fclose(fp);
}
```

Como pode ser observado no exemplo acima, normalmente, a função ***perror()*** é usada em conjunto com a macro ***ferror()***.

LEITURA COMPLEMENTAR

15. PONTEIROS

Sempre que utilizamos a função ***scanf()*** para ler o elemento de um vetor, passamos o endereço de memória deste elemento. Para isto, como já visto, utilizamos o operador **&**, o qual retorna o endereço de memória alocado para uma variável. Portanto, se ***v[i]*** **representa o i-ésimo elemento do vetor**, ***&v[i]*** **representa o endereço da memória em que esse elemento está armazenado**.

Na verdade, existe uma associação forte entre vetores e ponteiros. Na declaração:

int v[10];

a variável *v*, a qual representa o vetor, é uma constante que representa seu endereço inicial, ou seja, *v* sem indexação aponta para o primeiro elemento do vetor. Portanto, a linguagem C suporta aritmética de ponteiros, isto é, pode-se somar e subtrair ponteiros, desde que o valor resultante aponte para a área reservada para o vetor.

Se *p* representa um ponteiro para um inteiro, *p+1* representa um ponteiro para o próximo inteiro armazenado na memória. Este incremento equivale a somar 2 bytes (tamanho de um ***int***) no endereço de memória armazenado no ponteiro *p*. Com isso, em um vetor temos as seguintes equivalências:

v+0 aponta para o 1º elemento do vetor;
v+1 aponta para o 1º elemento do vetor;
v+2 aponta para o 1º elemento do vetor;
...
v+N aponta para o N-ésimo elemento do vetor;

Portanto, escrever ***&v[i]*** é equivalente a escrever ***(v + i)***. De maneira análoga, escrever ***v[i]*** é equivalente a escrever ****(v + i)***. Isto porque, o operador ***** retorna o valor contido na posição de memória indicada por ***(v + i)***.

No C a declaração de um ponteiro é feita pela seguinte sintaxe:

Tipo_dado *nome_variavel[tamanho1][tamanho2]...[tamanhoN];

A seguir é apresentado o exemplo de vetores reescrito para utilizar ponteiro.

```
#include <stdio.h>
#include <conio.h>
void main (void)
{
        int *num;  /* Declara um ponteiro para valores inteiros */
        int count=0;
        int totalnums;
        clrscr();
        do
        {
           printf ("Entre com um número não nulo ou digite 0 p/ terminar: ");
           scanf ("%d",num+count);
```

```
    count++;
  } while ((*(num+count-1) != 0) && (count < 100));
  totalnums=count-1;
  printf ("\nOs números digitados foram:\n");
  for (count=0; count <= totalnums; count++)
  {
    printf (" %d",*num);
    num++;
  }

}
```

Apesar da forma indexada ser mais clara, **o uso de ponteiros é muito importante,** principalmente por **possibilitar a alocação dinâmica de memória** (reservar espaço em memória para as variáveis durante a execução do programa). Apesar disso, tanto ponteiros quanto alocação dinâmica não fazem parte do escopo deste curso.

O C é altamente dependente dos ponteiros. Por isso, para ser um bom programador em C é fundamental que se tenha um bom domínio deles.

15.1. COMO FUNCIONAM OS PONTEIROS

O tipo ***int*** guardam inteiros. O tipo ***float*** guardam números de ponto flutuante. O tipo ***char*** guardam caracteres. **Ponteiros guardam endereços de memória**. Quando você anota o endereço de um colega você está criando um ponteiro. O ponteiro é este seu pedaço de papel. Ele tem anotado um endereço. Qual é o sentido disto? Simples. Quando você anota o endereço de um colega, depois você vai usar este endereço para achá-lo. O C funciona assim. Voce anota o endereço de algo numa variável ponteiro para depois usar.

Da mesma maneira, uma agenda, onde são guardados endereços de vários amigos, poderia ser vista como sendo uma matriz de ponteiros no C.

Um ponteiro também tem tipo. Veja: quando você anota um endereço de um amigo você o trata diferente de quando você anota o endereço de uma firma. Apesar de o endereço dos dois locais ter o mesmo formato (rua, número, bairro, cidade, etc.) eles indicam locais cujos conteúdos são diferentes. Então os dois endereços são ponteiros de tipos diferentes.

No C quando declaramos ponteiros nós informamos ao compilador para que tipo de variável vamos apontá-lo. Um ponteiro int aponta para um inteiro, isto é, guarda o endereço de um inteiro.

15.2. DECLARANDO E UTILIZANDO PONTEIROS

Para declarar um ponteiro temos a seguinte forma geral:

tipo_do_ponteiro *nome_da_variável;

É o operador asterisco (*****) que faz o compilador saber que aquela variável não vai guardar um valor, mas sim um endereço para aquele tipo especificado. Vamos ver exemplos de declarações:

```
int *pt;

char *temp,*pt2;
```

O primeiro exemplo declara um ponteiro para um inteiro. O segundo declara dois ponteiros para caracteres. Eles ainda não foram inicializados (como toda variável do C que é apenas declarada).

Isto significa que eles apontam para um lugar indefinido. Este lugar pode estar, por exemplo, na porção da memória reservada ao sistema operacional do computador. Usar o ponteiro nestas circunstânicias pode levar a um travamento do micro, ou a algo pior.

O ponteiro deve ser inicializado (apontado para algum lugar conhecido) antes de ser usado! Isto é de suma importância!

Para atribuir um valor a um ponteiro recém-criado poderíamos igualá-lo a um valor de memória. Mas, como saber a posição na memória de uma variável do nosso programa? Seria muito difícil saber o endereço de cada variável que usamos, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e realocados na execução. Podemos então deixar que o compilador faça este trabalho por nós. Para saber o endereço de uma variável basta usar o operador **&**. Veja o exemplo:

```
int count=10;

int *pt;

pt=&count;
```

Criamos um inteiro ***count*** com o valor 10 e um apontador para um inteiro ***pt***. A expressão ***&count*** nos dá o endereço de memória de ***count***, que é armazenado em ***pt***. Note que não alteramos o valor de ***count***, que continua valendo 10.

Como nós colocamos um endereço em ***pt***, ele está agora "liberado" para ser usado. Podemos, por exemplo, alterar o valor de ***count*** usando ***pt***. Para tanto vamos usar o operador "inverso" do operador **&**, ou seja, o operador *****. No exemplo acima, uma vez que fizemos ***pt=&count*** a expressão ****pt*** é equivalente ao próprio ***count***. Isto significa que, se quisermos mudar o valor de ***count*** para 12, basta fazer ****pt=12***.

Vamos fazer uma pausa e voltar à nossa analogia para ver o que está acontecendo.

Digamos que exista uma firma. Ela é como uma variável que já foi declarada. Você tem um papel em branco onde vai anotar o endereço da firma. O papel é um ponteiro do tipo firma. Você então liga para a firma e pede o seu endereço, o qual você vai anotar no papel. Isto é equivalente, no C, a associar o papel à firma com o operador **&**. Ou seja, o operador **&** aplicado à firma é equivalente a você ligar para a mesma e pedir o endereço. Uma vez de posse do endereço no papel você poderia, por exemplo, fazer uma visita à firma. No C você faz uma visita à firma aplicando o operador ***** ao papel. Uma vez dentro da firma você pode copiar seu conteúdo ou modificá-lo.

Uma observação importante: apesar do símbolo ser o mesmo, o operador ***** (**multiplicação**) não é o mesmo operador que o ***** (**referência de ponteiros**). Para começar **o primeiro é binário**, e o **segundo é unário pré-fixado**.

Aqui vão dois exemplos de usos simples de ponteiros:

```
#include <stdio.h>     /* Exemplo 1 */
int main ()
{
        int num,valor;
        int *p;
        num=55;
        p=&num;     /* Pega o endereco de num */
        valor=*p;       /* Valor e igualado a num de uma maneira indireta */
        printf ("\n\n%d\n",valor);
        printf ("Endereco para onde o ponteiro aponta: %p\n",p);
```

```
        printf ("Valor da variavel apontada: %d\n",*p);
        return(0);
}


#include <stdio.h>     /* Exemplo 2 */
int main ()
{
        int num,*p;
        num=55;
        p=&num;     /* Pega o endereco de num */
        printf ("\nValor inicial: %d\n",num);
        *p=100; /* Muda o valor de num de uma maneira indireta */
        printf ("\nValor final: %d\n",num);
        return(0);
}
```

No primeiro exemplo, o código ***%p*** usado na função ***printf()*** indica à função que ela deve imprimir um endereço.

Podemos fazer algumas operações aritméticas com ponteiros. A primeira, e mais simples, é igualar dois ponteiros. Se temos dois ponteiros ***p1*** e ***p2*** podemos igualá-los fazendo ***p1=p2***. Repare que estamos fazendo com que ***p1*** aponte para o mesmo lugar que ***p2***. Se quisermos que a variável apontada por ***p1*** tenha o mesmo conteúdo da variável apontada por ***p2*** devemos fazer ****p1=*p2***. Basicamente, depois que se aprende a usar os dois operadores (**&** e *****) fica fácil entender operações com ponteiros.

As próximas operações, também muito usadas, são o incremento e o decremento. Quando incrementamos um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta. Isto é, se temos um ponteiro para um inteiro e o incrementamos ele passa a apontar para o próximo inteiro. Esta é mais uma razão pela qual o compilador precisa saber o tipo de um ponteiro: se você incrementa um ponteiro ***char **** ele anda 1 byte na memória e se você incrementa um ponteiro double* ele anda 8 bytes na memória. O decremento funciona semelhantemente. Supondo que p é um ponteiro, as operações são escritas como:

```
p++;
p--;
```

Mais uma vez insisto. Estamos falando de operações com ponteiros e não de operações com o conteúdo das variáveis para as quais eles apontam. Por exemplo, para incrementar o conteúdo da variável apontada pelo ponteiro p, faz-se:

```
(*p)++;
```

Outras operações aritméticas úteis são a soma e subtração de inteiros com ponteiros. Vamos supor que você queira incrementar um ponteiro de 15. Basta fazer:

```
p=p+15;  ou  p+=15;
```

E se você quiser usar o conteúdo do ponteiro 15 posições adiante:

```
*(p+15);
```

A subtração funciona da mesma maneira. Uma outra operação, às vezes útil, é a comparação entre dois ponteiros. Mas que informação recebemos quando comparamos dois ponteiros? Bem,

em primeiro lugar, podemos saber se dois ponteiros são iguais ou diferentes (== e !=). No caso de operações do tipo >, <, >= e <= estamos comparando qual ponteiro aponta para uma posição mais alta na memória. Então uma comparação entre ponteiros pode nos dizer qual dos dois está "mais adiante" na memória. A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer:

p1>p2

Há entretanto operações que você não pode efetuar num ponteiro. Você não pode dividir ou multiplicar ponteiros, adicionar dois ponteiros, adicionar ou subtrair float ou double de ponteiros.

15.3. PONTEIROS E VETORES

VETORES COMO PONTEIROS

Quando você declara uma matriz da seguinte forma:

tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];

o compilador C calcula o tamanho, em bytes, necessário para armazenar esta matriz. Este tamanho é:

tam1 x tam2 x tam3 x ... x tamN x tamanho_do_tipo

O compilador então aloca este número de bytes em um espaço livre de memória. O nome da variável que você declarou é na verdade um ponteiro para o tipo da variável da matriz. Este conceito é fundamental. Eis porque: Tendo alocado na memória o espaço para a matriz, ele toma o nome da variável (que é um ponteiro) e aponta para o primeiro elemento da matriz.

Mas aí surge a pergunta: então como é que podemos usar a seguinte notação?

nome_da_variável[índice]

Isto pode ser facilmente explicado desde que você entenda que a notação acima é absolutamente equivalente a se fazer:

****(nome_da_variável+índice)***

Agora podemos entender como é que funciona um vetor! Vamos ver o que podemos tirar de informação deste fato. Fica claro, por exemplo, porque é que, no C, a indexação começa com zero. É porque, ao pegarmos o valor do primeiro elemento de um vetor, queremos, de fato, ****nome_da_variável*** e então devemos ter um índice igual a zero. Então sabemos que:

****nome_da_variável é equivalente a nome_da_variável[0]***

Outra coisa: apesar de, na maioria dos casos, não fazer muito sentido, poderíamos ter índices negativos. Estaríamos pegando posições de memória antes do vetor. Isto explica também porque o C não verifica a validade dos índices. Ele não sabe o tamanho do vetor. Ele apenas aloca a memória, ajusta o ponteiro do nome do vetor para o início do mesmo e, quando você usa os índices, encontra os elementos requisitados.

Vamos ver agora um dos usos mais importantes dos ponteiros: a varredura sequencial de uma matriz. Quando temos que varrer todos os elementos de uma matriz de uma forma sequencial, podemos usar um ponteiro, o qual vamos incrementando. Qual a vantagem? Considere o seguinte programa para zerar uma matriz:

```
int main ()
```

```
{
        float matrx [50][50];
        int i,j;
        for (i=0;i<50;i++)
        for (j=0;j<50;j++)
                matrx[i][j]=0.0;
        return(0);
}
```

Podemos reescrevê-lo usando ponteiros:

```
int main ()
{
        float matrx [50][50];
        float *p;
        int count;
        p=matrx[0];
        for (count=0;count<2500;count++)
      {
        *p=0.0;
        p++;
      }
        return(0);
}
```

No primeiro programa, cada vez que se faz ***matrx[i][j]*** o programa tem que calcular o deslocamento para dar ao ponteiro. Ou seja, o programa tem que calcular 2500 deslocamentos. No segundo programa o único cálculo que deve ser feito é o de um incremento de ponteiro. Fazer 2500 incrementos em um ponteiro é muito mais rápido que calcular 2500 deslocamentos completos.

Há uma diferença entre o nome de um vetor e um ponteiro que deve ser frisada: um ponteiro é uma variável, mas o nome de um vetor não é uma variável. Isto significa, que não se consegue alterar o endereço que é apontado pelo "nome do vetor". Seja:

```
int vetor[10];
int *ponteiro, i;
ponteiro = &i;

/* as operacoes a seguir sao invalidas */

vetor = vetor + 2;     /* ERRADO: vetor nao e' variavel */
vetor++;               /* ERRADO: vetor nao e' variavel */
vetor = ponteiro;      /* ERRADO: vetor nao e' variavel */
```

Teste as operações acima no seu compilador. Ele dará uma mensagem de erro. Alguns compiladores dirão que vetor não é um ***Lvalue*** ("***Left value***"). Isto significa que, um símbolo que pode ser colocado do lado esquerdo de uma expressão de atribuição, ou seja, uma variável. Outros compiladores dirão que tem-se "***incompatible types in assignment***", tipos incompatíveis em uma atribuição.

```
/* as operacoes abaixo sao validas */
```

```
ponteiro = vetor;      /* CERTO: ponteiro e' variavel */
ponteiro = vetor+2;    /* CERTO: ponteiro e' variavel */
```

PONTEIROS COMO VETORES

Sabemos agora que, na verdade, o nome de um vetor é um ponteiro constante. Sabemos também que podemos indexar o nome de um vetor. Como consequência podemos também indexar um ponteiro qualquer. O programa mostrado a seguir funciona perfeitamente:

```
#include <stdio.h>
int main ()
{
        int matrx [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int *p;
        p=matrx;
        printf ("O terceiro elemento do vetor e: %d",p[2]);
        return(0);
}
```

Podemos ver que ***p[2]*** equivale a ****(p+2)***.

ENDEREÇOS DE ELEMENTOS DE VETORES

Nesta seção vamos apenas ressaltar que a notação

&nome_da_variável[índice]

é válida e retorna o endereço do ponto do vetor indexado por índice. Isto seria equivalente a ***nome_da_variável + indice***. É interessante notar que, como consequência, o ponteiro ***nome_da_variável*** tem o endereço ***&nome_da_variável[0]***, que indica onde na memória está guardado o valor do primeiro elemento do vetor.

VETORES DE PONTEIROS

Podemos construir vetores de ponteiros como declaramos vetores de qualquer outro tipo. Uma declaração de um vetor de ponteiros inteiros poderia ser:

int *pmatrx [10];

No caso acima, pmatrx é um vetor que armazena 10 ponteiros para inteiros.

15.4. INICIALIZANDO PONTEIROS

Podemos inicializar ponteiros. Vamos ver um caso interessante dessa inicialização de ponteiros com strings.

Precisamos, para isto, entender como o C trata as strings constantes. Toda string que o programador insere no programa é colocada num banco de strings que o compilador cria. No local onde está uma string no programa, o compilador coloca o endereço do início daquela ***string*** (que está no banco de strings). É por isto que podemos usar ***strcpy()*** do seguinte modo:

strcpy (string,"String constante.");

strcpy() pede dois parâmetros do tipo ***char****. Como o compilador substitui a string "String constante." pelo seu endereço no banco de strings, tudo está bem para a função ***strcpy()***.

O que isto tem a ver com a inicialização de ponteiros? É que, para uma string que vamos usar várias vezes, podemos fazer:

char *str1="String constante.";

Aí poderíamos, em todo lugar que precisarmos da string, usar a variável ***str1***. Devemos apenas tomar cuidado ao usar este ponteiro. Se o alterarmos vamos perder a string. Se o usarmos para alterar a string podemos facilmente corromper o banco de strings que o compilador criou.

Mais uma vez fica o aviso: ponteiros são poderosos, mas, se usados com descuido, podem ser uma ótima fonte de dores de cabeça.

15.5. PONTEIROS PARA PONTEIROS

Um ponteiro para um ponteiro é como se você anotasse o endereço de um papel que tem o endereço da casa do seu amigo. Podemos declarar um ponteiro para um ponteiro com a seguinte notação:

tipo_da_variável **nome_da_variável;

Algumas considerações: *****nome_da_variável*** é o conteúdo final da variável apontada; ****nome_da_variável*** é o conteúdo do ponteiro intermediário.

No C, podemos declarar vários níveis de aninhamentos de ponteiros para ponteiros, ou seja, ponteiros para ponteiros; ponteiros para ponteiros para ponteiros; e assim por diante. Para fazer isto basta aumentar o número de asteriscos na declaracão.

Para acessar o valor desejado apontado por um ponteiro para ponteiro, o operador asterisco deve ser aplicado duas vezes, como mostrado no exemplo abaixo:

```
#include <stdio.h>
int main()
{
   float fpi = 3.1415, *pf, **ppf;
   pf = &fpi;          /* pf armazena o endereco de fpi */
   ppf = &pf;          /* ppf armazena o endereco de pf */
   printf("%f", **ppf);  /* Imprime o valor de fpi */
   printf("%f", *pf);    /* Tambem imprime o valor de fpi */
   return(0);
}
```

15.6. CUIDADOS A SEREM TOMADOS NA UTILIZAÇÃO DE PONTEIROS

O principal cuidado ao se usar um ponteiro deve ser: **saiba sempre para onde o ponteiro está apontando**. Portanto, **nunca use um ponteiro que não foi inicializado**. Um pequeno programa que demonstra como não usar um ponteiro:

```
int main () /* Errado - Nao Execute */
{
        int x,*p;
        x=13;
        *p=x;
        return(0);
```

```
}
```

Este programa compilará e rodará. O que acontecerá? Ninguém sabe. O ponteiro p pode estar apontando para qualquer lugar. Você estará gravando o número 13 em um lugar desconhecido. Com um número apenas, você provavelmente não vai ver nenhum defeito. Agora, se você começar a gravar números em posições aleatórias no seu computador, não vai demorar muito para travar o micro (se não acontecer coisa pior).

16. ALOCAÇÃO DINÂMICA

A alocação dinâmica permite ao programador alocar memória para variáveis quando o programa está sendo executado. Assim, pode-se definir, por exemplo, um vetor ou uma matriz cujo tamanho é definido em tempo de execução.

Existem diversas outras funções que são amplamente utilizadas, mas dependentes do ambiente e compilador. No entanto, **o padrão ANSI C define apenas quatro funções para o sistema de alocação dinâmica,** disponíveis no arquivo de cabeçalho ***stdlib.h***, e descritas a seguir.

16.1. FUNÇÃO MALLOC()

A função ***malloc()*** serve para alocar um espaço de memória. Sua sintaxe geral é:

ptr = malloc (num);

Esta funçao aloca na memória o número de bytes definido por ***num***. Ela retorna um ponteiro ***ptr*** do tipo ***void ** para o primeiro byte alocado. **Este ponteiro pode ser atribuído a qualquer tipo de ponteiro**. Se não houver memória suficiente para a alocação requisitada, a função **retorna um ponteiro nulo** (**NULL**).

Veja um exemplo de alocação dinâmica com ***malloc()***:

```
#include <stdio.h>
#include <stdlib.h>    /* Para usar malloc() */
void main (void)
{
        int *p;
        int a, i;
        a = 100;
        /* O comando abaixo atribui ao ponteiro p o espaço necessário para guardar a Nº
        inteiros, assim, p agora pode ser tratado como um vetor com 100 posições */
        p = (int *) malloc (a*sizeof(int) );
        if (!p)
        {
                printf ("** Erro: Memoria Insuficiente **");
                exit;
        }
        for (i = 0; i < a ; i++)
                p[i] = i * i;
}
```

No exemplo acima, é alocada memória suficiente para se armazenar ***a*** números inteiros. O operador ***sizeof()*** retorna o número de bytes de um **inteiro**. Ele é util para se saber o tamanho de tipos. O **ponteiro *void**** que ***malloc()*** retorna é convertido para um ***int**** pelo operador ***cast*** e é

atribuído a ***p***. O comando seguinte testa se a operação foi bem sucedida. Se não tiver sido, ***p*** terá um valor nulo, portanto, **!p** será verdadeiro. Se a operação tiver sido bem sucedida, podemos usar o vetor de inteiros alocados normalmente, por exemplo, indexando-o de ***p[0]*** a ***p[(a-1)]***.

16.2. FUNÇÃO CALLOC()

A função ***calloc()*** também serve para alocar memória, mas sua sintaxe é um pouco diferente:

ptr = calloc (num, size);

Esta funçao aloca uma quantidade de memória igual a ***num * size***, ou seja, **aloca memória suficiente para um vetor de *num* objetos de tamanho *size***. Assim como no caso anterior, ***ptr*** também é um ponteiro do tipo ***void ** para o primeiro byte alocado. Se não houver memória suficiente para alocar a memória requisitada a função ***calloc()*** retorna um ponteiro nulo.

Portanto, o programa anterior pode ser rescrito para realizar a alocação dinâmica com ***calloc()***:

```
#include <stdio.h>
#include <stdlib.h>    /* Para usar calloc() */
void main (void)
{
        int *p;
        int a, i;
        a = 100;
        p=(int *)calloc(a,sizeof(int));
        if (!p)
        {
                printf ("** Erro: Memoria Insuficiente **");
                exit;
        }
        for (i = 0; i < a ; i++)
                p[i] = i * i;
}
```

16.3. FUNÇÃO REALLOC()

A função ***realloc()*** serve para redefinir a alocação de memória. Sua sintaxe geral é:

ptr = realloc (ptr, num);

Esta funçao modifica o tamanho da memória previamente alocada e apontada por ***ptr*** para aquele especificado por ***num***. O valor de ***num*** pode ser maior ou menor que o tamanho original. Um ponteiro para a nova alocação é devolvido. Isto ocorre porque ***realloc()*** pode precisar mover o bloco alocado originalmente para aumentar seu tamanho, ou seja, alterar o endereço de memória dos bytes alocados. Se isso ocorrer, o conteúdo do bloco antigo é copiado no novo bloco, e **nenhuma informação é perdida**. Se ***ptr*** for nulo, aloca ***num*** bytes e devolve um ponteiro. Se ***num*** é zero, a memória apontada por ***ptr*** é liberada. **Se não houver memória suficiente para a realocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado**.

No programa abaixo, a alocação existente no ponteiro ***p***, realizada pela da função ***malloc()***, é alterada através da função ***realloc()***.

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

```
#include <stdlib.h>    /* Para usar malloc()  e realloc()*/
void main (void) {
        int *p;
        int a, i;
        a = 30;
        /* No comando abaixo, p recebe o espaço necessário para guardar 30 Nº inteiros */
        p=(int *)malloc(a*sizeof(int));
        if (!p) {
                printf ("** Erro: Memoria Insuficiente **");
                exit; }
        for (i=0; i<a ; i++)
                p[i] = i*i;
        a = 80;
        /* O comando abaixo, realocará o tamanho de p a partir do novo valor de a (80) */
        p = realloc (p, a*sizeof(int));
        for (i=0; i<a ; i++)
                p[i] = a*i*(i-6); }
```

16.4. FUNÇÃO FREE()

Quando um espaço de memória alocado dinamicamente não é mais necessário, o mesmo deve ser liberado (desalocado). Para isto, existe a função ***free()*** cuja sintaxe é:

```
free (ptr);
```

Onde ***ptr*** é o ponteiro que aponta para o início da memória alocada.

Através deste ponteiro, o programa busca numa "***tabela de alocação interna***" a quantidade de bytes que devem ser liberados.

No exemplo abaixo, o espaço alocado para p pela função ***malloc()*** é liberado pela função ***free()*** ao final do programa:

```
#include <stdio.h>
#include <stdlib.h>    /* Para usar malloc() e free() */
void main (void) {
        int *p;
        int a, i;
        a = 100;
        p = (int *) malloc (a*sizeof(int) );
        if (!p) {
                printf ("** Erro: Memoria Insuficiente **");
                exit;  }
        for (i = 0; i < a ; i++)   {
                p[i] = i * i;
                printf ("\nO elemento %d possui o valor %d.", i, p[i]);  }
        /* O comando abaixo libera o espaço de memória alocado para o ponteiro p */
        free(p);
}
```