PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
DIRETORIA DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO
DISCIPLINA DE ASPECTOS FORMAIS DA COMPUTAÇÃO
PROFESSOR CAMILLO OLIVEIRA

ASPECTOS FORMAIS

Curitiba, 1999.

SUMÁRIO

pag
1 – Introdução 5
1.1 – Origens da linguagem C 5
1.2 – Esqueleto de um programa em linguagem C 7
1.3 – Mapa de memória de um programa em C 8
2 – Tipos básicos 8
3 – Operador de atribuição 10
4 – Constantes 11
4.1 – Constantes caracteres 11
4.2 – Constantes inteiras 11
4.3 – Constantes com ponto flutuante 11
4.4 – Constantes octais e hexadecimais 11
4.5 – Constantes string 11
4.6 – Constantes caractere de barra invertida 11
5 – Conversão de tipos 12
5.1 – Exercícios propostos 13
6 – Operadores 14
6.1 – Operadores aritméticos 14
6.1.1 – Incremento e decremento 15
6.1.2 – Expressões aritméticas 16
6.2 – Operadores lógicos e relacionais 17
6.2.1 – Operadores Lógicos 17
6.2.1.1 – Tabelas verdade 17
6.2.2 – Operadores Relacionais 17
6.2.3 – Expressões lógicas 17
6.3 – Operadores de bits 18
6.4 – Precedência dos operadores 19
6.5 – Abreviaturas 20
7 – Variáveis compostas homogêneas 20
7.1 – Variáveis compostas homogêneas unidimensionais (vetores) 20
7.1.1 – Inicialização de vetores 22
7.2 – Variáveis compostas homogêneas multidimensionais (matrizes) 23
7.2.1 – Inicialização de matrizes 24
7.3 – Exercícios propostos 26
7.4 – Operador em tempo de compilação **sizeof** 32
8 – Ponteiros 32
8.1 – Variáveis ponteiros 33
8.2 – Operadores de ponteiros 33
8.3 – Aritmética de ponteiros 33
8.4 – Indireção múltipla 33
8.5 – Ponteiros e strings 34
8.6 – Ponteiros e vetores 35
8.7 – Ponteiros e matrizes 35
8.8 – Vetor de ponteiros 36
8.9 – Exercícios propostos 37
9 – Variáveis compostas heterogêneas 38
9.1 – Estruturas 38
9.2 – Exercícios propostos 40
9.3 – Uniões 62
10 – Enumerações 64
11 – Entradas e saídas utilizando scanf(...) e printf(...) 64
11.1 – printf(...) 64
11.2 – scanf(...) 65
11.3 – Exercícios propostos 66

12 – Comandos de controle do fluxo de execução 69
12.1 – Comandos de seleção 69
12.1.1 – Comando **if** 69
12.1.1.1 – Eros mais comuns 75
12.1.1.2 – Operador de seleção (**? :**) 76
12.1.2 – Comando **switch** 78
12.1.3 – Exercícios propostos 79
12.2 – Comandos de repetição 82
12.2.1 – Comando **while** 82
12.2.2 – Comando **do** - **while** 83
12.2.3 – Comando **for** 85
12.3 – Comandos de desvios incondicionais 88
12.3.1 – Comando **break** 88
12.3.2 – Comando **continue** 88
12.4 – Exercícios propostos 89
12.4.1 – Séries 89
12.4.1.1 – Respostas dos exercícios sobre séries 90
12.4.2 – Vetores 91
12.4.3 – Matrizes 92
12.4.4 – Exercícios gerais sobre repetições 95
13 – Funções 97
13.1 – Passagem de parâmetros por valor 99
13.2 – Passagem de parâmetros por referência 99
13.3 – Chamando funções, passando e retornando estruturas 102
13.4 – Chamando funções, passando um vetor como parâmetro 104
13.5 – Chamando funções, passando uma matriz como parâmetro 107
14 – Alocação dinâmica de memória 109
15 – Exercícios propostos 110
15.1 – Strings 110
15.2 – Operadores de bits 115
16 – Arquivos 121
16.1 – Representação de registros 121
16.2 – Fluxos e arquivos 122
16.3 – fopen(...) 124
16.4 – fread(...) 124
16.5 – fwrite(...) 125
16.6 – fclose(...) 126
16.7 – ftell(...) 126
16.8 – fseek(...) 127
16.9 – feof(...) 128
16.10 – fgets(...) e fputs(...) 128
16.11 – Exercícios propostos 130
17 – Glossário 134
18 – Bibliografia 136

1. INTRODUÇÃO

1.1 ORIGEM DA LINGUAGEM C

A linguagem C é resultado de uma linguagem mais antiga chamada BCPL, que originou a linguagem B (Ken Thompson), que por sua vez influenciou ao aparecimento da linguagem C (Dennis Ritchie), nos laboratórios BELL em 1972, tornando-se a linguagem básica do sistema operacional UNIX.

Por muito tempo, a linguagem C padrão foi fornecida junto com o sistema operacional UNIX. Com a popularidade dos microcomputadores, a linguagem C passou a ser usada com bastante intensidade. Foi quando em 1983, um comitê ANSI (AMERICAN NATIONAL STANDARDS INSTITUTE) padronizou a linguagem C com 32 palavras chaves (27 originais e mais 5 incorporadas pelo comitê). Para efeito de comparação, a linguagem BASIC para o IBM-PC, possui 159 palavras chaves.

A linguagem C combina o controle e as estruturas de dados de uma linguagem de alto nível com a facilidade de trabalhar em equipamentos (máquinas) a um nível associado e aproximado com a linguagem de montagem (assembler). Possui uma sintaxe concisa bastante atrativa para os programadores, sendo que os compiladores geram um código objeto muito eficaz.

As linguagens de alto nível, são linguagens ricas em tipos de dados. São os tipos de dados que definem um conjunto de valores de uma variável pode armazenar e o conjunto de operações que podem ser executados com esta variável.

Linguagens de <u>ALTO NÍVEL</u>:
- ADA
- MODULA - 2
- PASCAL
- COBOL
- FORTRAN
- BASIC
- VISUAL BASIC

Linguagem de <u>MÉDIO NÍVEL</u>:
- C
- MACRO ASSEMBLER

Linguagem de <u>BAIXO NÍVEL</u>:
- ASSEMBLER

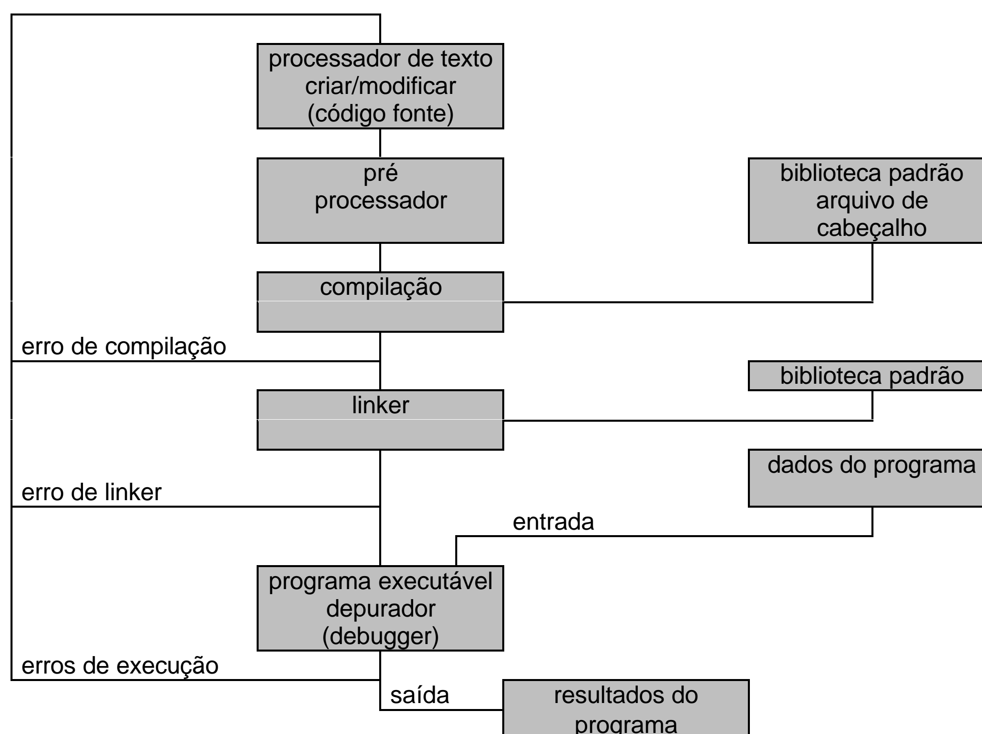
A linguagem C é uma linguagem estruturada, que permite diversas estruturas de laços (repetições) como: **while**, **do - while** e **for**.

A linguagem C suporta o conceito de bloco de código, que é um conjunto de instruções entre chaves, Exemplo:

```
if (x < 10)
{
	printf("Muito baixo, tente novamente\n");
	scanf("%d",&x);
}
```

A linguagem C é dita para programadores, ao contrário de outras linguagem, as de <u>ALTO NÍVEL</u>, o programador não necessita de conhecimento em informática para programá-las. A linguagem C foi criada, influenciada e testada em campo por programadores profissionais. O resultado final é que a linguagem C dá ao programador o que ele quer: poucas restrições, poucas reclamações, estruturas de blocos, funções isoladas e um conjunto compacto de palavras chaves. Usando a linguagem C, um programador pode conseguir aproximadamente a eficiência de código de montagem (máquina, assembler). Não é de admirar que a linguagem C seja tranqüilamente a linguagem mais popular entre os programadores profissionais.

O fato da linguagem C, freqüentemente ser usada em lugar da linguagem Assembly é o fator mais importante para a sua popularidade entre os programadores. A linguagem Assembly usa uma representação simbólica do código binário real que o computador executa diretamente. Cada operação em linguagem Assembly leva a uma tarefa simples a ser executada pelo computador. Embora a linguagem Assembly dê aos programadores o potencial de realizar tarefas com máxima flexibilidade e eficiência, é notoriamente difícil de trabalhar quando se está desenvolvendo ou depurando um programa. Além disso, como a linguagem Assembly não é uma linguagem estruturada, o programa final tende a ser um emaranhado de jumps, calls e índices (desvios). Essa falta de estrutura torna os programas em linguagem Assembly difíceis de ler, aperfeiçoar e fazer manutenções necessárias a qualquer programa de computador. Talvez mais importantes: as rotinas em linguagem Assembly não são portáveis entre máquinas com unidades centrais de processamento (UCP's) diferentes.



Digrama mostra a seqüência de eventos necessários para a implementação de um programa em linguagem C.

Processador de texto - é usado para criar o texto do programa (código fonte) ou modificar o texto já existente.

Pré processadores - o programa inicia com o pré processamento de diretivas que ativam à substituição de macros, compilações condicionais e a inclusão de outros arquivos textos (código fonte) no programa. Exemplo: arquivos de cabeçalho (header files) que suprem detalhes sobre funções das bibliotecas. O pré processador lê um arquivo texto (código fonte), expande alguma macro, incluir algum arquivo de cabeçalho (header files) e escreve o resultado para um arquivo texto intermediário que é lido pelo compilador.

Compilador - o compilador converte o arquivo gerado pelo pré processador em um código objeto, que contém código executável de máquina, mais informações para o linker. Exemplo: detalhes de funções padrões requeridas. Alguns compiladores podem gerar uma listagem combinada entre o compilador e o assembler, que apresenta o código original do C e as sentenças em linguagem assembler.

Linker - Exceto para programas muito especialistas (programas embutidos), a maioria dos programas requer funções da biblioteca padrão. O linkeditor junta os arquivos de código objeto, bibliotecas padrão e outros arquivos gerados pelo usuário, para formar o arquivo executável (em código de máquina para ser executado sob o sistema operacional.

Execução do programa - O usuário através do prompt do sistema operacional, pode executar o programa. Durante a execução do programa, dados podem ser entrados via teclado e ou arquivos e resultados podem sair para o monitor de vídeo, arquivos e ou impressoras.

Depuradores (debugger) - muitos ambientes de programação possuem um depurador automático que ajuda na detecção e correção de erros de compilação, de linkedição (junção), erros de execução e erros de lógica de programação, estes dois últimos erros só aparecem durante a execução do programa. Nestes depuradores, pontos de quebra podem ser inseridos (significa que você pode executar o programa até o ponto marcado como quebra), examinar o conteúdo de variáveis etc. Quando o programa trabalha corretamente (foi corretamente implementado), o depurador não é mais necessário e o programa roda (executa) por si só.

Os depuradores geralmente são interpretadores. Um interpretador executa o código fonte uma linha por vez, executando à instrução específica contida nessa linha. Por isso um programa interpretado é mais lento que o compilado.

Um ambiente integrado de desenvolvimento contém um processador de texto, habilitando o programador a escrever o código fonte e pressionando uma tecla pré definida ou o botão do mouse, ativa o processo de compilação e linkedição. Se, em tempo de compilação e ou linkedição, ocorrer erros, o programa para a compilação e mostra as mensagens de erro e o usuário deve corrigi-las. Se a compilação e a linkedição completar e sem erros, o programa executável foi gerado podendo ser executado dentro do ambiente (sob o controle do depurador) ou fora, direto no sistema operacional.

Um erro pode ocorrer:

durante a compilação (em tempo de compilação): erro de sintaxe, que indica que o programa não está escrito conforme as regras da linguagem, exemplo: a estrutura de uma sentença está incorreta (erro de sintaxe).

durante a linkedição (em tempo de linkedição): múltiplas definições de uma referência (duas funções com o mesmo nome) ou uma referência indefinida (uma função chamada no programa e não encontrada).

durante a execução (em tempo de execução): o programa aborta com um erro de run time (divisão por zero) ou não faz o que deve ser feito (resultado incorreto).

1.2 ESQUELETO DE UM PROGRAMA EM LINGUAGEM C

```
/* declaração das variáveis globais */

void main()
{       /* início de bloco */

        /* declaração das variáveis */

        /* seqüência de comando e funções */
}       /* final de bloco */
```

Sempre um programa ou um conjunto de programas em linguagem C, deve ter uma única função **main**. Sempre aparecerá um destes protótipos: **void main(), void main(void), int main(), int main(void), main() ou main(void)**. Exemplo de um programa bem simples seria:

Para exemplificar os processos descritos anteriormente, observe o código abaixo:

```
#include <stdio.h>          // pré processador de diretivas

void main()         // cabeçalho (header) da função main
{       // início do bloco de código da função main
        puts("Engenharia de computação . . . ");
}       // final do bloco de código da função main.
```

1.3 MAPA DE MEMÓRIA EM C

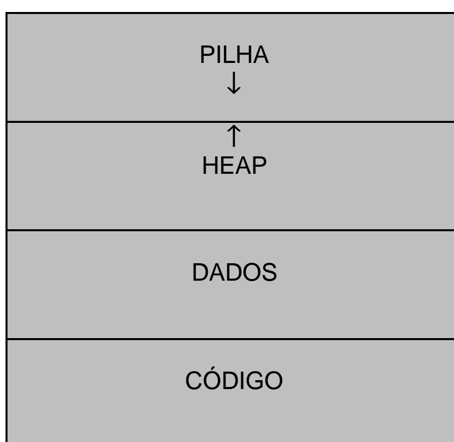
Um programa compilado em linguagem C, cria e usa quatro regiões, que são: área de código, área de dados, heap e pilha.

A pilha tem diversos usos durante a execução de seu programa. Ela possui o endereço de retorno das chamadas de funções, argumentos para as funções e variáveis locais, também guarda o estado atual da UCP (unidade central de processamento).

O heap é uma região de memória livre, que seu programa pode usar nas funções de alocações dinâmicas em linguagem C, em aplicações como listas encadeadas e árvores.

As áreas do heap e pilha são ditas áreas dinâmicas, por que são utilizadas (manipuladas) durante a execução do programa (em tempo de execução).

A disposição exata do programa pode variar de compilador para compilador e de ambiente para ambiente. Por isso é dado um modelo conceitual da memória.



2. TIPOS BÁSICOS

Na linguagem C existem cinco tipos de dados, que são: caractere (**char**), inteiro (**int**), ponto flutuante (**float**), ponto flutuante de precisão dupla (**double**) e sem valor (**void**). Veremos que todos os outros tipos de dados são baseados em um desses tipos. 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.

O formato exato de valores em ponto flutuante depende de como estão implementados. Inteiros geralmente correspondem ao tamanho natural da palavra do computador hospedeiro. Valores caracteres são geralmente usados para conter valores definidos pelo conjunto de caracteres ASCII (AMERICAN NATIONAL STANDARD CODE for INFORMATION INTERCHANGE). Os valores fora desta faixa podem ser manipulados diferentemente entre as implementações em C.

Um programa de computador manipula dados, os dados tem que ser armazenados em algum lugar. Este lugar é chamado de memória. Qualquer linguagem de programação deve oferecer meios de acesso à memória, ou seja, um meio de reservar um pedaço da memória e um meio de identificar (dar um nome) a este pedaço que foi reservado de memória.

A declaração de uma variável, reserva um espaço na memória onde será guardado um tipo de dado especificado.

O tipo do dado informa ao compilador quanto de espaço de memória é necessário para armazenar aquele tipo de dado e como o dado deve ser manipulado. Por exemplo: para o computador e muito diferente a soma de dois números inteiros e dois números reais.

Na linguagem C a declaração de variáveis obedece a seguinte sintaxe:

[modificador] tipo identificador;
[modificador] tipo identificador 1, identificador 2, . . . ;

Os tipos básicos da linguagem C são: **char** (caractere), **int** (inteiro), **float** (real), **double** (real) e **void** (sem valor).

Os modificadores do tipo (opcional) são: **signed** (sinalizado, com sinal), **unsigned** (não sinalizado, sem sinal), **long** (longo) e **short** (curto). Os modificadores são opcionais na declaração da variável. Exceto o tipo **void**, os demais tipos de dados básicos podem ter vários modificadores precedendo-os. Um modificador é usado para alterar o significado de um tipo básico para adaptá-lo mais precisamente às necessidades de diversas situações.

Os modificadores **signed**, **short**, **long** e **unsigned** podem ser aplicados aos tipos básicos caractere e inteiro. O modificador **long** também pode ser aplicado ao tipo básico **double**. (Observe que o padrão ANSI elimina o **long double** porque ele tem o mesmo significado de um **double**.

O identificador é uma seqüência de caracteres alfanuméricos e o caracter _ (sublinhado). O primeiro caracter da seqüência deve ser uma letra ou um sublinhado e os caracteres subseqüentes devem ser letras, números ou sublinhados. O padrão ANSI C define os nomes de variáveis, funções, rótulos (labels) e vários outros objetos definidos pelo usuário como identificadores.

Exemplo:

Correto	Incorreto
contador	1contador
teste344	teste!344
_ano	%_ano
tempo_médio	tempo...médio

Exemplo de declarações de variáveis

float x, salário, nota, media;
unsigned int ano;
char c, sexo;
double x1, y1, z1, razão, sal_inicial, sal_final;

A tabela a seguir mostra o tamanho e o intervalo numérico em uma arquitetura de 16 bits.

Tipo	**Tamanho**		**Intervalo**
	Bytes	**Bits**	
char **signed char**	1	8	-128 a 127
unsigned char	1	8	0 a 255
short int **signed short int**	2	16	-32.768 a 32.767
unsigned short int	2	16	0 a 65.535
int **signed int**	2	16	-32.768 a 32.767
unsigned int	2	16	0 a 65.535
long int **signed long int**	4	32	-2.147.483.648 a 2.147.483.647
unsigned long int	4	32	0 a 4.294.967.295
float	4	32	3.4E-38 a 3.4E+38 (7 dígitos de precisão)
double	8	64	1.7E-308 a 1.7E+308 (15 dígitos de precisão)
long double	10	80	3.4E-4932 a 1.1E+4932

Para uma arquitetura de 32 bits, o que muda é o tamanho do inteiro (**int**), que é do tamanho da palavra (32 bits). O **short int** passa a ser 16 bits e o **long int** 64 bits.

Obs. Vale lembrar que quando declaramos uma variável, um espaço (em bytes) na memória do computador é reservado para a variável.

Obs. O padrão ANSI determina que os identificadores podem ter qualquer tamanho, mas pelo menos os primeiros seis caracteres devem ser significativos se o identificador estiver envolvido em um processo externo de linkedição. São chamados então de **nome externos**. Caso contrário são chamados de **nomes internos** e os 31 primeiros caracteres são significativos. É bom olhar o manual do usuário para ver exatamente quantos caracteres significativos são permitidos pelo compilador que se está usando.

Obs. Letras maiúsculas e minúsculas são tratadas diferentemente. Portanto ano, Ano, ANO são três identificadores distintos.

Obs. Um identificador não pode ser igual à um palavra chave da linguagem C e não deve ter o mesmo nome que as funções que você escreveu ou as que estão na biblioteca.

3. OPERADOR DE ATRIBUIÇÃO

Na linguagem C, você pode usar o operador de atribuição dentro de qualquer expressão válida em C. Isto não acontece com a maioria das linguagem de computador.

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

        a = 2 + (b = 3);
}
```

Analise o programa acima e informe os valores das variáveis a e b.

a = _________________ b = _________________

Quando quisermos atribuir um valor a uma determinada posição de memória utilizaremos o símbolo **=**, que costumamos chamar de igual.

O operando a esquerda (*lvalue*) sempre se refere a uma posição de memória, o operando a direita (*rvalue*) será um valor.

A sintaxe seria: *nome_da_variável = expressão;*

```
int a, b;

a = 5;     // o valor 5 vai para a posição de memória da variável a
b = a;     // o valor armazenado na variável a é copiado para b
```

A atribuição de um valor inicial para uma variável pode ser feita na mesma hora em que é feita a sua declaração.

```
unsigned long int x = 10, b = 200;
```

Para toda variável global é atribuído um valor inicial igual à zero (nulo), o que não ocorre com uma variável local, que sempre possui sujeira da memória (um valor que não interessa, por isso é chamado de lixo).

```
a = b + 1 = c = 2 + 5; // erro, b + 1 retorna um valor, não um lvalue
```

4. CONSTANTES

Referem-se aos valores fixos que o programa não pode alterar. Estas constantes podem ser de qualquer um dos cinco tipos de dados básico.

4.1 CONSTANTES CARACTERES

São envolvidas por aspas simples. Um caractere entre apóstrofos (aspas simples), ‘a’, é uma constante que tem o valor do código ASCII que a representa.

unsigned char a = ‘B’; // a possui o valor do código ASCII que representa (66).
char x = ‘%’; // x possui o valor do código ASCII.

4.2 CONSTANTES INTEIRAS

Uma constante inteira é representada com números sem componentes fracionários. O tipo de uma constante inteira é o menor tipo de dado capaz de armazená-la: **int**, **unsigned int**, **long int**, **unsigned long int**. A menos que a variável venha acompanhada de um sufixo que indique o seu tipo: **U - unsigned int**, **L- long int** e **UL - unsigned long int**.

4.3 CONSTANTES COM PONTO FLUTUANTE (NÚMEROS REAIS)

Constantes com ponto flutuante requerem ponto decimal seguido pela parte fracionária do número. Existem dois tipos básicos de variáveis com ponto flutuante: **float** e **double**. Podemos também utilizar sufixos que indicam o seu tipo: **F - float** e **L - long double**.

Tipos de dados	Exemplo de constantes
int	1 123 21000 -234
long int	35000L -34L
short int	10 -12 90
unsigned int	10000U 977U 40000U
float	123.23F 4.34e-3F 124.F 2e1
double	123.23 12312333. -0.9876432
long double	101.2L

4.4 CONSTANTES HEXADECIMAIS E OCTAIS

Muitas vezes são utilizados variáveis no sistema numérico na base 8 (octal) e na base 16 (hexadecimal). O sistema octal utiliza os dígitos 0 a 7 e o hexadecimal os dígitos 0 a 9 e as letras A, B, C, D, E e F para representar os algarismos 10, 11, 12, 13, 14 e 15.

Uma constante hexadecimal deve consistir de um 0x seguido por uma constante na forma hexadecimal. Uma constante octal com um 0 seguido por uma constante na forma octal.

int h = 0xF   /* 15 em decimal */
int 0 = 012   /* 10 em decimal */

4.5 CONSTANTES STRING

São constantes formadas por um conjunto de caracteres (bytes) entre aspas “ (aspas duplas). Por exemplo “PUC-Pr.”.

4.6 CONSTANTES CARACTERE DE BARRA INVERTIDA

São constantes especiais para os caracteres de controle como o **return**, **line feed**, **beep** e etc.. Você deve utilizar os códigos de barra invertida em lugar de seu ASCII equivalente para aumentar a portabilidade.

Código	Significado
\b	Retrocesso (BS)
\f	Alimentação de formulário (FF)
\n	Nova linha
\r	Retorno de carro (CR)
\t	Tabulação horizontal (HT)
\”	“
\’	‘
\0	Nulo
\\	Barra invertida
\v	Tabulação vertical
\a	Alerta (beep)
\N	Constante octal (onde N é uma constante octal
\xN	Constante hexadecimal (onde N é uma constante hexadecimal)

5. CONVERSÃO DE TIPOS

A conversão de tipos refere-se à situação em que variáveis de um tipo são misturadas com variáveis de outro tipo. Em um comando de atribuição, a regra de conversão de tipos é muito simples: **o valor do lado direito** (o lado da expressão) **de uma atribuição é convertido no tipo do dado esquerdo** (a variável destino).

Quando se converte de inteiros para caracteres, inteiros longos para inteiros, a regra básica é que a quantidade apropriada de bits significativos será ignorada. Isso significa que 8 bits são perdidos quando se vai de inteiro para caractere ou inteiro curto, e 16 bits são perdidos quando se vai de um inteiro longo para um inteiro.

A conversão de um **int** em um **float** ou **float** em **double** etc. não aumenta a precisão ou exatidão. Esses tipos de conversão apenas mudam a forma em que o valor é representado.

Tipos destino	Tipo da expressão	Possível informação perdida
char	**unsigned char**	Se valor > 127, o destino é negativo
char	**int**	Os 8 bits mais significativos
char	**unsigned int**	Os 8 bits mais significativos
char	**long int**	Os 24 bits mais significativos
char	**unsigned long int**	Os 24 bits mais significativos
int	**long int**	Os 16 bits mais significativos
int	**unsigned long int**	Os 16 bits mais significativos
int	**float**	A parte fracionária e as vezes mais
int	**double**	A parte fracionária e as vezes mais
float	**double**	Precisão, o resultado é arredondado
double	**long double**	Precisão, o resultado é arredondado

5.1 EXERCÍCIOS PROPOSTOS

1. Dado o programa

```
void main()
{
        char x, a;
        unsigned char b;
        int y, z;
        unsigned long int w;

        x = 0x7F;
        w = 65535U;
        y = 1439;
        x = x + 1;
        a = b = y;
        z = w;
}
```

Analise o programa anterior e informe o conteúdo das variáveis x, y, z, a, b e w, quando a execução do programa estiver na linha 14. (dê o resultados em valores na base 10)

x = __________ y = __________ z = __________

a = __________ b = __________ w = __________

2. Analise o programa que segue, executando-o até a linha 10 e explique o conteúdo das variáveis x, y e z.

```
void main()
{
        double x;
        float y;
        int z;

        x = 2.1234567890123456789012345;
        y = x;
        z = y;
}
```

x = __________ y = __________ z = __________

Justifique a resposta:

____
____
____
____
____

3. Analise o programa que segue, informe e justifique o conteúdo das variáveis a e b, quando a execução do programa estiver na linha 8.

```
void main()
{
        char a;
        int b;

        a = 66;
        b = 'B';
}
```

a = ____________ b = ____________

Justifique a resposta (analise a tabela ASCII):

____
____
____
____
____

4. Analise o programa que segue, informe o conteúdo das variáveis r, s, t, u e v, quando a execução do programa estiver na linha 14. ***Sugestão: observe as variáveis no watch em hexadecimal***.

```
void main()
{
        unsgined long int r;
        int s;
        unsigned int t;
        char u;
        unsigned char v;

        r = 0x12345678L;
        s = r;
        t = r;
        u = r;
        v = r;
}
```

r = __________ s = __________ t = __________

u = __________ v = __________

6. OPERADORES

A linguagem C é rica em operadores internos. A linguagem C dá mais ênfase aos operadores que a maioria das outras linguagens de computador. Na linguagem C podemos definir operadores: aritméticos, relacionais, lógicos e de bits. Além disso outros operadores utilizados para tarefas particulares.

Os operadores geram dados intermediários, que podem ser utilizados.

6.1 OPERADORES ARITMÉTICOS

Operadores aritméticos	
-	Subtração, também menos unário
+	Adição
*	Multiplicação
/	Divisão
%	Módulo da divisão (resto)
--	Decremento
++	Incremento

Exemplo:

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

        a = 5 / 2;
        b = -a;
}
```

Quando o operador **/** (divisão) for utilizado a um inteiro ou caractere, qualquer resto é truncado. Por isso na linha 5, após a sua execução, o valor armazenado na variável **a** será o valor 2, visto que é uma divisão de inteiros (não suporta à parte fracionária).

O menos unário (linha 6) multiplica seu único operando por -1. Isso é, qualquer número precedido por um sinal de menos troca-se o sinal. Portanto o valor armazenado na variável **b** na linha 6 seria o valor -2.

6.1.1 INCREMENTO E DECREMENTO

A linguagem C possui dois operadores úteis geralmente não encontrados em outras linguagens. São os operadores de incremento e decremento, ++ e --. O operador ++ soma 1 ao seu operando, e -- subtrai 1.

```
x = x + 1;        // é o mesmo que ++x ou x++;
x = x - 1;// é o mesmo que --x ou x--;
```

Os dois tipos de operadores de incremento e decremento podem ser utilizados como prefixo ou sufixo do operando (++c, c++, --y, y--).

Os operadores de incremento e decremento operam sobre um lvalue e retornam ao valor armazenado na variável.

Exemplo:

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

        a = 5;
        c = ++a; // incrementa a em 1 e atribui o valor de a para a                // posição
de memória c
        ++a++;  // erro pois o resultado intermediário gerado por ++a              // é um valor, não um
lvalue. Não pode ser usado                          // como operando para o operador de
incremento.
}
```

Exemplo:

```
void main()
{
        int a, b, c, d, x = 10, y = 5;

        a = x++;
        b = ++x;
        c = --y;
        d = y--;
}
```

Analisando o programa anterior, notamos que na linguagem C, valores podem ser atribuídos as variáveis na hora da declaração das mesmas (linha 3). Assim para as variáveis **x** e **y** são atribuídos os valores iniciais de 10 e 5 respectivamente.

Quando um operador de incremento ou decremento precede seu operando, C executa a operação de incremento ou decremento antes de usar o valor do operando. Se o operador estiver após seu operando, C usa o valor do operando antes de incrementá-lo ou decrementá-lo.

Fazendo um teste de mesa para o programa anterior teríamos:

a	b	c	d	x	y
10	12	4	4	10 11 12	5 4 3

A precedência dos operadores aritméticos seria:

++ --
* / %
+ -

Os operadores do mesmo nível de precedência são avaliados pelo computador da esquerda para a direita. Obviamente, parênteses podem ser usados para alterar a ordem de avaliação. A linguagem C trata os parênteses da mesma forma que todas as outras linguagens de programação. Parênteses forçam uma operação, ou um conjunto de operações, a ter um nível de precedência maior.

6.1.2 EXPRESSÕES ARITMÉTICAS

São expressões onde os operandos podem ser constantes e ou variáveis e os operadores são aritméticos.

Exemplo:

```
void main()
{
        int cnt, acm, ind1, ind2;

        cnt = 100;
        acm = 100 + cnt;
        ind1 = acm + cnt++;
        ind2 = acm + ++cnt;
        cnt = --ind1 + 1000;
}
```

Mostrando o teste de mesa do programa acima temos os seguintes valores atribuídos para as variáveis ao longo da execução do programa.

cnt	acm	ind1	ind2
100 101 102 1299	200	300 299	302

O programa anterior poderia ser escrito da seguinte maneira:

```
void main()
{
        int cnt, acm, ind1, ind2;

        cnt = 100;
        acm = 100 + cnt;
        ind1 = acm + cnt;
        cnt++;
        ++cnt;
        ind2 = acm + cnt;
        --ind1;
        cnt = ind1 + 1000;
}
```

OBS. Sempre que tivermos dúvidas com operadores ++ e ou --, é uma boa tática explodirmos (expandírmos) a expressão em mais linhas de código, de modo que consigamos observar melhor a seqüência de operações a serem feitas.

6.2 OPERADORES LÓGICOS E RELACIONAIS

6.2.1 OPERADORES LÓGICOS

<table>
<tr><th>Operador</th><th>Ação</th><th></th></tr>
<tr><td>&&</td><td>and</td><td>Conjunção</td></tr>
<tr><td>||</td><td>or</td><td>Disjunção</td></tr>
<tr><td>!</td><td>not</td><td>negação</td></tr>
</table>

6.2.1.1 TABELAS VERDADE

<table>
<tr><th colspan="3">&&</th></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
</table>

<table>
<tr><th colspan="3">||</th></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
</table>

<table>
<tr><th colspan="2">!</th></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td></tr>
</table>

Perceba que não temos o operador lógico **xor** na linguagem C, então, devemos obtê-lo a partir de uma expressão lógica utilizando **and**, **or** ou **not**.

6.2.2 OPERADORES RELACIONAIS

Operador	Ação
>	maior que
>=	maior ou igual à
<	menor que
<=	menor ou igual à
==	igual
!=	diferente

O importante, nos operadores lógicos e relacionais, é que estes são utilizados em expressões lógicas e o resultado de uma expressão lógica é **0** (falso) ou **1** (verdadeiro).

As precedências entre os operadores lógicos e relacionais seriam:

<table>
<tr><td>!</td></tr>
<tr><td>> >= < <=</td></tr>
<tr><td>== !=</td></tr>
<tr><td>&&</td></tr>
<tr><td>||</td></tr>
</table>

6.2.3 EXPRESSÕES LÓGICAS

São expressões onde os operando são constantes e ou variáveis e os operadores são lógicos e ou relacionais. O resultado de uma expressão lógica é sempre **1** (um) para verdadeiro ou **0** (zero) para falso.

As expressões lógicas serão utilizadas nos comandos de seleção (**if**) e repetição (**while, do while, for**), estas expressões é que controlarão o fluxo de execução do programa através dos comandos. Mais na frente veremos com mais detalhes estes comandos.

Exemplos de expressões lógicas:

```
Altura < 1.80 && Peso < 100
(x < 10 && y > 10) && z == 10
Salário < 2000.00 || Horas > 40
```

6.3 OPERADORES DE BITS

Ao contrário de muitas linguagens, a linguagem C suporta um completo conjunto de operadores de bits. Uma vez que C foi projetada para substituir a linguagem assembly na maioria das tarefas de programação, era importante dar suporte as operações que podem ser feitas em linguagem assembly. Operações com bits consistem em testar, atribuir ou deslocar os bits efetivos em um byte ou palavra, que correspondem aos tipos **char**, **int** ou variantes do padrão C. Operações com bits não podem ser usadas em **float**, **double**, **long double**, **void** ou outros tipos mais complexos.

Os operadores de bits são:

Operador	Ação
&	and
|	or
^	xor
~	complemento
>>	deslocamento. para a direita
<<	deslocamento para a esquerda

Os operadores de bits encontram aplicações mais freqüentemente em "drivers" de dispositivos - como em programas de modems, rotinas de arquivos em disco e rotinas de impressoras. - porque as operações com bits mascaram certos bits.

Para ligar um determinado bit, ou seja, torná-lo **1**, utiliza-se o operador |. Na seqüência temos um exemplo que ilustra

```
 0000000000011000001
 0001000000010100000  |
---------------------
 0001000000011100001
```

Para desligar um bit, ou seja, torná-lo **0** (zero), utiliza-se o operador &. Na seqüência temos um exemplo para desligar um bit.

```
 0100100000011000001
 1111011111111111111  &
---------------------
 0100000000011000001
```

O operador ^ geralmente é utilizado para inverter o estado de um bit.

```
 0100100000011000001
 0000100000000000000  ^
---------------------
 0100000000011000001
```

O operador de complemento **~**, inverte o estado de cada bit da variável especificada. Ou seja, todo **1** vira **0** e todo **0** vira **1**.

Os operadores de bits são usados freqüentemente em rotinas de criptografia. Se você deseja fazer um arquivo em disco parecer ilegível, realize algumas manipulações de bits no arquivo. Um dos métodos mais simples é complementar cada byte usando o complemento de um para inverter cada bit no byte. Veja o exemplo que segue:

0100100010011011001 ~
1011011101100100110 1º complemento
0100100010011011001 2º complemento

Para finalizar os operadores de bits, temos os deslocamento de bit para a direita (**>>**) e para a esquerda (**<<**).

Suponha **x** ser uma variável do tipo **unsigned char**.

Antes		Ação		Depois
X = 7		00000111		7
X <<= 1	0	00001110	←	14
X <<= 3	000	01110000	←	112
X <<= 2	00001	11000000	←	192
X >>= 1	→	01100000	0	96
X >>= 2	→	00011000	00	24

Quando temos: **x <<= 1;**, é o mesmo que **x = x << 1;**, ou seja, os bits de **x** serão deslocados para a esquerda em 1 e o resultado do deslocamento é atribuído na variável **x**. Este tipo de notação vale para todos os operadores de bits.

Vale a pena observar que o deslocamento em 1 para a esquerda, você multiplica o número por 2 e o deslocamento em 1 para a direita é o mesmo que dividir o número por 2.

6.4 PRECEDÊNCIA DOS OPERADORES

<table>
<tr><th colspan="2">Precedência</th><th colspan="2">Operadores</th><th>Associatividade</th></tr>
<tr><td>Mais alta</td><td>15</td><td></td><td>. -> () []</td><td>esquerda para direita</td></tr>
<tr><td></td><td>15</td><td>unário</td><td>lvalue++lvalue--</td><td>direita para esquerda</td></tr>
<tr><td></td><td>14</td><td>unário</td><td>! ~ + - ++lvalue --lvalue</td><td>direita para esquerda</td></tr>
<tr><td></td><td>14</td><td>unário</td><td>(typecast) sizeof</td><td>direita para esquerda</td></tr>
<tr><td></td><td>13</td><td>multiplicativo.</td><td>* / %</td><td>esquerda para direita</td></tr>
<tr><td></td><td>12</td><td>aditivo</td><td>+ -</td><td>esquerda para direita</td></tr>
<tr><td></td><td>11</td><td>deslocamento</td><td><< >></td><td>esquerda para direita</td></tr>
<tr><td></td><td>10</td><td>relacional</td><td>< <= > >=</td><td>esquerda para direita</td></tr>
<tr><td></td><td>9</td><td>igualdade</td><td>== !=</td><td>esquerda para direita</td></tr>
<tr><td></td><td>8</td><td>bit a bit AND</td><td>&</td><td>esquerda para direita</td></tr>
<tr><td></td><td>7</td><td>bit a bit XOR</td><td>^</td><td>esquerda para direita</td></tr>
<tr><td></td><td>6</td><td>bit a bit OR</td><td>|</td><td>esquerda para direita</td></tr>
<tr><td></td><td>5</td><td>lógico AND</td><td>&&</td><td>esquerda para direita</td></tr>
<tr><td></td><td>4</td><td>lógico OR</td><td>||</td><td>esquerda para direita</td></tr>
<tr><td></td><td>3</td><td>condicional</td><td>? :</td><td>direita para esquerda</td></tr>
<tr><td></td><td>2</td><td>atribuição</td><td>= *= /= %= += -= >>= <<= &= ^= |=</td><td>direita para esquerda</td></tr>
<tr><td>mais baixa</td><td>1</td><td>seqüência</td><td>,</td><td>esquerda para direita</td></tr>
</table>

Obs. Vale lembrar que parênteses e colchetes afetam a ordem de avaliação da expressão.

6.5 ABREVIAÇÕES (ABREVIATURAS) NA LINGUAGEM C

A linguagem C oferece abreviações (abreviaturas) que simplificam a codificação de certos tipos de comandos de atribuição.

x = x + 10;

Significa pegar o conteúdo da variável **x**, somar dez (10) e atribuir o resultado na variável **x**. Isto poderia ser escrito de outra forma, produzindo o mesmo resultado, que seria:

x += 10;

O compilador entende atribuir à variável **x** o valor da variável **x** mais 10, que é mesmo resultado produzido anteriormente.

Esta mesma idéia vale para os operadores de bits e matemáticos, como segue:

Forma abreviada	Forma não abreviada
x += 3;	x = x + 3;
x -= 4;	x = x - 4;
x /= 10;	x = x / 10;
x %= 2;	x = x % 2;
x *= 5.6;	x = x * 5.6
x &= 2;	x = x & 2;
x |= 5;	x = x | 5;
x ^= 2;	x = x ^2;
x <<= 3;	x = x << 3;
x >>= 2;	x = x >> 2

É importante saber sobre as abreviações (abreviaturas), visto que nos livros e programas feitos por programadores profissionais, você sempre irá encontrar a forma abreviada.

7. VARIÁVEIS COMPOSTAS HOMOGÊNEAS

Do mesmo modo que na teoria de conjuntos, uma variável pode ser interpretada como um elemento e uma estrutura de dados como um conjunto. Quando uma determinada estrutura de dados for composta de variáveis com o mesmo tipo primitivo, temos um conjunto homogêneo de dados.

7.1 VARIÁVEIS COMPOSTAS UNIDIMENSIONAIS (VETORES)

Para entendermos variáveis compostas unidimensionais, imaginemos um edifício com um número finito de andares, representando uma estrutura de dados, e seus andares, partições dessa estrutura. Visto que os andares são uma segmentação direta do prédio, estes compõem então o que é chamado de estrutura unidimensional (uma dimensão).

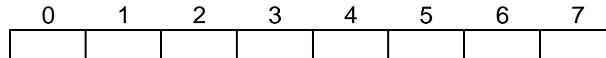
Um vetor é uma coleção de variáveis do mesmo tipo que são referenciadas por um nome comum. Na linguagem C, todos os vetores consistem em localizações contíguas de memória.

A declaração de um vetor seria:

[modificador] tipo identificador[*tamanho*]*;*

Obs. Os colchetes do identificador, delimitando o tamanho do vetor, não são opcionais.

unsigned int vet[8];



```
void main()
{
        int v[5];

        v[0] = 2;
        v[1] = 10;
        v[2] = v[0] + 5 * v[1];
        v[4] = v[2] - 8 / v[0];
        v[3] = v[1];
}
```

Mostrando o teste de mesa do programa acima temos:

v[0]	v[1]	v[2]	v[3]	v[4]	5 * v[1]	8 / v[0]
2	10	52	10	48	50	4

O programa abaixo mostra que: os índices de um vetor pode ser uma variável ou uma expressão. Note que o operador ++ está depois do operando **i** (sufixo). Analise os valores finais das variáveis.

```
void main(void)
{
        int v[5], i;

        i = 0;
        v[i++] = 0x2;
        v[i++] = 10;
        v[i++] = v[0] + 5 * v[1];
        v[i++] = v[2] - 8 / v[0];
        v[i++] = v[1];
}
```

O programa abaixo possui um erro de programação que ocorrerá em tempo de execução. Qual o erro que o programa tem? Faça um teste de mesa e justifique a sua resposta.

```
void main(void)
{
        int v[5], i;

        i = 0;
        v[++i] = 0x2;
        v[++i] = 10;
        v[++i] = v[0] + 5 * v[1];
        v[++i] = v[2] - 8 / v[0];
        v[++i] = v[1];
}
```

____
____
____
____
____

7.1.1 INICIALIZAÇÃO DE VETORES

Um vetor pode ser inicializado logo após a sua declaração, como segue:

unsigned long int v[4] = {100,1234,1200,1233};

0	1	2	3
100	1234	1200	1233

float b[4] = {2.4,124,2e1,3E-5};

0	1	2	3
2.4	124	2e1	3E-5

double tipos[4] = {-0.02,1.24,3,-5.6};

0	1	2	3
-0.02	1.24	3	-5.6

Ao inicializarmos um vetor de caracteres, temos:

char nome[14] = "Computação";

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
C	o	m	p	u	t	a	ç	ã	o	\0				

char nome[10] = "Computação";

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
C	o	m	p	u	t	a	ç	ã	o					

unsigned char x[7] = "PUC-PR";

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P	U	C	-	P	R	\0								

unsigned char y[9] = {'P', 'U' , 'C' , '-' , 'P' , 'R', '\0'};

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P	U	C	-	P	R	'\0'								

char var[5] = "P";

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P	\0													

Faça um teste de mesa e explique o conteúdo da variável str, quando a execução do programa estiver na linha 10.

```
1.  void main()
2.  {
3.      int x;
4.      unsigned char str[16] = "ASPECTOS";

5.      x = 040;
6.      str[0] ^= x;
7.      str[1] ^= x;
8.      str[2] ^= x;
9.      str[3] ^= x;
10. }
```

str = ____________________

Justifique a resposta (observe a tabela de códigos ASCII):
____
____
____
____
____

OBSERVAÇÕES SOBRE VETORES

O cálculo do tamanho em bytes de um vetor está diretamente em função do seu tamanho e o tipo básico. Que seria: **total em bytes = tamanho do tipo * tamanho do vetor**.

Na linguagem C não existe uma verificação dos limites dos vetores. Pode-se ultrapassar os limites e escrever dados no lugar de outra variável ou mesmo no código do programa. É responsabilidade do programador prover à verificação dos limites onde for necessário, de modo nunca invadir espaço na memória, de uma outra variável.

7.2 VARIÁVEIS COMPOSTAS MULTIDIMENSIONAIS (MATRIZES)

Ainda no exemplo do edifício, suponha que além do acesso pelo elevador até um determinado andar, tenhamos também a divisão do andar em apartamentos. Para chegar a algum deles não basta só o número do andar, precisamos também do número do apartamento.

As estruturas compostas unidimensionais tem como principal característica a necessidade de apenas um índice para o endereçamento - são estrutura com uma única dimensão. Uma estrutura que precise de mais de um índice, como no caso do edifício dividido em apartamentos, seria então denominada estrutura composta multidimensional (neste caso duas dimensões (bidimensional)).

Imagine um conjunto de três prédios de apartamentos. Teríamos uma estrutura homogênea tridimensional, com três índices. Para chegarmos ao apartamento desejado, os índices seriam: o número do prédio, o andar e o número do apartamento.

Dependendo do problema que se tem para resolver, podemos pensar na estrutura multidimensional que quisermos.

A declaração de uma matriz seria:

[modificador] tipo identificador[*num. de linhas*][*num. de colunas*]*;*

Quando declaramos uma matriz, um espaço, em bytes, é reservado na memória e a matriz se encontra linearmente a partir de um endereço base.

Exemplo:

double m[5][5];

	0	1	2	3	4
0					
1					
2					
3					
4					

7.2.1 INICIALIZAÇÃO DE MATRIZES

Podemos inicializarmos uma matriz logo após a sua declaração, como segue:

unsigned long int v[2][2] = {{1000,1234},{1200,1233}};

	0	1
0	1000	1234
1	1200	1233

float b[3][2] = {{2.4,124},{2e1,3E-5},{-0.11,2.}};

	1	2
0	2.4	124
1	2e1	3E-5
2	-0.11	2.

double matriz[2][10] ={{1,1,1,1,1,1,1,1,1,1},{2,2,2,2,2,2,2,2,2,2}};

	0	1	2	3	4	5	6	7	8	9
0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0

Exemplo:

```
1.  void main()
2.  {
3.      int x, y;
4.      double m[3][4] = {{0,0,0,0},{0,0,0,0},{0,0,0,0}};
5.
6.      x = 2;
7.      y = 3;
8.      m[0][0] = 10;
9.      m[2][1] = 4.34 + m[x - 2][y - 3];
10.     m[x][x] = 3.e-3;
11.     m[x][y] = 100.01;
12. }
```

Teríamos a seguinte matriz quando a execução do programa estivesse na linha 12.

	0	1	2	3
0	10.0	0.0	0.0	0.0
1	0.0	3.e-3	0.0	0.0
2	0.0	14.34	3.e-3	100.01

char Nomes[3][15] = {"Carla","Manoel","Maria"};

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	C	a	r	l	a	\0									
1	M	a	n	o	e	l	\0								
2	M	a	r	i	a	\0									

Podem existir aplicações que utilizem matrizes que possuam mais que duas dimensões. Um exemplo seria:

int m[3][2][4] = {{{4,4,4,4},{4,4,4,4}},{{5,5,5,5},{5,5,5,5}},{{6,6,6,6},{6,6,6,6}}};

Esquematicamente ficaria:

```
      2 5666
    1 5555 6
  0 4444 5
    4444
```

OBSERVAÇÕES SOBRE MATRIZES

O cálculo do tamanho em bytes de uma matriz está diretamente relacionado com o seu tamanho e o tipo básico. Que seria: **total em bytes = tamanho do tipo * número de linhas * número de colunas**.

Na linguagem C não existe uma verificação dos limites das matrizes. Pode-se ultrapassar os limites e escrever dados no lugar de outra variável ou mesmo no código do programa. É responsabilidade do programador prover à verificação dos limites onde for necessário, de modo nunca invadir o espaço na memória, de uma outra variável.

7.3 EXERCÍCIOS PROPOSTOS

1. Analise o programa abaixo, informando o conteúdo das variáveis a e b, quando a execução estiver na linha 11.

```
1.   void main()
2.   {
3.        unsigned int a;
4.        unsigned int b;

5.        a = 0x9129;
6.        b = 0x8218;
7.        a >>= 4;
8.        a &= 0x00FF;
9.        b >>= 4;
10.       b &= 0x00FF;
11.  }
```

a = ____________ b = ____________

2. Analise o código que segue e informe o conteúdo das variáveis a e b, quando a execução estiver na linha 9.

```
1.   void main()
2.   {
3.        unsigned char a[3];
4.        char b[3] = {0x80,0x09,0xB0};

5.        a[0] = b[0];
6.        b[1] = ~b[1];
7.        a[1] = b[1];
8.        a[2] = b[2];
9.   }
```

a = ____________________b = ____________________

3. Analise o programa abaixo, informando o conteúdo das variáveis a, b e c, quando a execução estiver na linha 15.

```
1.   void main()
2.   {
3.        unsigned char a[5];
4.        char b[5];

5.        c = 0x3FF;
6.        a[0] = b[0] = c;
7.        c <<= 1;
8.        a[1] = b[1] = c;
9.        c ^= 0x80;
10.       a[2] = b[2] = c;
11.       c >>= 4;
12.       a[3] = b[3] = c;
13.       c |= 0x90;
14.       a[4] = b[4] = c;
15.  }
```

a = ____________________b = ____________________c = _____

4. Dado o código:

```
1.    void main()
2.    {
3.          unsigned char a;
4.          char b;
5.          unsigned int c;
6.          int d;

7.          a = b = c = d = 0x0FF;
8.          c = d |= 0x8000;
9.    }
```

Qual o conteúdo das variáveis a, b, c e d, quando a execução do programa estiver na linha 9.

a = ____________ b = ____________

c = ____________ d = ____________

5. Analise o código abaixo e informe o conteúdo da variável v, quando a execução estiver na linha 9.

```
1.    void main(void)
2.    {
3.          int v[4], v1, v2;

4.          v1 = v2 = 1;
5.          v[0] = 0xF & 032 + 1;
6.          v[1] = 2 * 0x80 + 02;
7.          v[2] = 1 + v2 * 2 >= v1 + 3 && v1;
8.          v[3] = (1 + v2) * 2 >= v1 + 3 && v1;
9.    }
```

6. Analise o programa abaixo e fazendo um teste de mesa e informe o conteúdo da variável mat, i e j, quando a execução do programa estiver na linha 12.

```
1.    void main()
2.    {
3.          int m[2][2];
4.          unsigned int i, j;

5.          m[0][0] = 1;
6.          m[0][1] = m[1][0] = m[1][1] = 0;
7.          i = j = 1;
8.          m[0][0] += 10;
9.          m[0][j] = m[i - 1][j - 1] + 10;
10.         m[i][0] = m[0][j] * (m[0][0] - 1);
11.         m[i][j] = m[0][0] - m[0][1] - m[1][0];
12.   }
```

Teste de mesa

m[0][0]	m[0][1]	m[1][0]	m[1][1]	i	j

m = ________________________ i = _____ j = _____

7. Analise o programa abaixo e informe o conteúdo da variável s, quando a execução do programa estiver na linha 7. ***Sugestão: utilize uma tabela de códigos ASCII para verificar o valor do caracteres e vice versa***.

```
1.  void main()
2.  {
3.      char s[5] = “Puc”;

4.      s[0] = ~s[0];
5.      s[1] = ~s[1];
6.      s[2] = ~s[2];
7.  }
```

s = ________________

8. Analise o programa abaixo e informe o conteúdo das variáveis x e y, quando a execução do programa estiver na linha 16.

```
1.  void main()
2.  {
3.      unsigned int n;
4.      char y[4];
5.      unsigned char x[4];
6.      unsigned long int a;

7.      n = 1;
8.      n %= 8;
9.      n++;
10.     a = 0xFFEEDDCCUL;
11.     x[0] = y[0] = n | a;
12.     x[1] = y[1] = a >> (n * 2);
13.     x[2] = y[2] = a >> (n * 4);
14.     x[3] = y[3] = a >> (n * 6);
15.     x[1] ^= y[2] ^= 0x80;
16. }
```

x = ___________________ y = ___________________

9. Analise o código abaixo e informe o conteúdo das variáveis x, y, z e w, quando a execução do programa estiver na linha 13.

```
1.  void main()
2.  {
3.      unsigned int n;
4.      unsigned char x, w;
5.      char y, z;

6.      n = 0x000A;
7.      x = 0xFFF0;
8.      z = 0x007F;
9.      x += (n % 0x0A);
10.     y = x;
11.     y += y;
12.     w = z = (z + 0x0001);
13. }
```

x = __________ y = __________

z = __________ w = __________

10. Analise o código que segue e informe o conteúdo da variável a, quando a execução do programa estiver na linha 15.

```
1.  void main()
2.  {
3.      unsigned int n;
4.      unsigned char a[5] = {0xFF,0xFE,0xFD,0xFC,0xFB};
5.      char i;

6.      n = 20;
7.      n %= 3;
8.      n += 2;
9.      i = -1;
10.     a[0] = a[++i] + n;
11.     a[1] = a[++i] + n;
12.     a[2] = a[i + 1] + n;
13.     a[3] = a[i + 2] + n;
14.     a[4] = a[i + 3] + n;
15. }
```

a = ______________________________

11. Analise o programa abaixo e informe o conteúdo da variável a, quando a execução do programa estiver na linha 19.

```
1.  void main()
2.  {
3.      unsgned int n;
4.      int a[7] = {07,07,07,07,07,07,07};
5.      double c, d;

6.      c = 1.1;
7.      d = 2.2;
8.      n >>= 4;
9.      n = 0x00F0;
10.     n %= 0x000F;
11.     a[4] = n;
12.     a[0] = (a[0] > n && n < 100) || n == 9;
13.     a[1] = a[0]++ + --a[2] - -a[3] + -a[4];
14.     a[2] = a[1] + a[2] > n || !(a[0] + 2 < n);
15.     a[3] = a[5]++ - ++a[6];
16.     a[4] = c + 1 > d;
17.     a[5] = ++a[0] + a[1]++;
18.     a[6] = --a[0] + -a[1];
19. }
```

a = ____

12. Analise o código que segue e informe o conteúdo das variáveis a e d, quando a execução do programa estiver na linha 12.

```
1.  void main()
2.  {
3.      int a[4], b, c, n;
4.      double d[4];

5.      n = 11;
6.      b = 10 * (n % 5) + 1;
7.      c = 10;
8.      a[0] = d[0] = b / c;
9.      a[1] = d[1] = b / (float) c;
10.     a[2] = d[2] = b / 20;
11.     a[3] = d[3] = b / 20.0;
12. }
```

a = ___________________________________

d = ___________________________________

13. Analise o código abaixo e informe o conteúdo das variáveis a, b e c, quando a execução do programa estiver na linha 11.

```
1.  void main()
2.  {
3.      int a[4] = {1,1,1,1}, b, c, n;

4.      n = 0x000B;
5.      b = n % 8;
6.      c = 2;
7.      a[0] = b++ - c;
8.      a[1] = ++b - c;
9.      a[2] += --a[0] + a[1]++;
10.     a[3] = ++a[0] - a[1] + a[2];
11. }
```

a = _____________________________________

b = _______________   c = _______________

14. Analise o código que segue e informe o conteúdo das variáveis a e b, quando a execução do programa estiver na linha 10.

```
1.  void main()
2.  {
3.      unsigned char a;
4.      char b, n;

5.      n = 0x14;
6.      n %= 0x10;
7.      n++;
8.      n |= 0x80;
9.      a = b = n;
10. }
```

a = _______________   b = _______________

15. Execute, o programa abaixo, até a linha 10 e informe o conteúdo das variáveis a e b.

```
1.  void main()
2.  {
3.      unsigned char a, b, c;
4.      char d, e, n;

5.      n = n % 0x10;
6.      a = 0x10;
7.      b = d = n - a;
8.      c = 0xFE;
9.      e = c;
10. }
```

a = _______________ b = _______________

16. Execute, o programa abaixo, até a linha 13 e informe o conteúdo da variável m.

```
1.  void main()
2.  {
3.      double x, m[3][2] = {{0,0},{0,0},{0,0}};
4.      int a, b;

5.      a = 1;
6.      b = 2;
7.      x = 3;
8.      m[a][a] = x / b;
9.      m[0][a] = 5 / b;
10.     a = b;
11.     m[a][b - a] = 7 / m[1][1];
12.     m[--a][0] = !((a + b) >= x);
13. }
```

m = ____

7.4 OPERADOR EM TEMPO DE COMPILAÇÃO SIZEOF(...)

O operador **sizeof** é um operador em tempo de compilação, unário, que retorna o tamanho, em bytes da variável ou especificador de tipo entre parênteses, que ele precede.

Analise o código abaixo, e informe quais os valores que estão no vetor y, quando a execução do programa estiver na linha 16.

y = ____

```
1.  void main()
2.  {
3.      unsigned int x, y[9];
4.      double a, b[6];
5.      char s, r[6] = “casa”;

6.      y[0] = sizeof(x);
7.      y[1] = sizeof(y);
8.      y[2] = sizeof(a);
9.      y[3] = sizeof(b);
10.     y[4] = sizeof(s);
11.     y[5] = sizeof(r);
12.     y[6] = sizeof(y) / sizeof(int);       /* número de elementos de y */
13.     y[7] = sizeof(b) / sizeof(double);    /* número de elementos de b */
14.     y[8] = sizeof(r) / sizeof(char);      /* número de elementos de r */
15.     x = y[0] + y[1] + y[2] + y[3] + y[4] + y[5] + y[6] + y[7] + y[8];
16. }
```

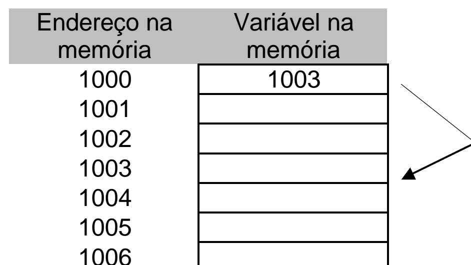
Por que os valores de y[1] e y[3] são 18 e 48 respectivamente?

____
____
____
____

O operador **sizeof** ajuda basicamente a gerar códigos portáveis que dependam do tamanho dos tipos de dados internos da linguagem C. Por exemplo, imagine um programa de banco de dados que precise armazenar seis valores inteiros por registro. Se você quer transportar o programa de banco de dados para vários computadores, não deve assumir o tamanho de um inteiro, mas deve determinar o tamanho do inteiro usando **sizeof**.

8. PONTEIROS

Ponteiro é uma variável que contém um endereço de memória. Esse endereço é normalmente a posição de uma outra variável na memória.



8.1 VARIÁVEIS PONTEIROS

Na linguagem C a declaração de variáveis ponteiros obedece a seguinte sintaxe:

*[modificador] tipo *identificador;*
*[modificador] tipo *identificador 1, *identificador 2, . . .;*

Onde o tipo básico pode ser qualquer tipo válido em C. O tipo define o tipo base do ponteiro, que indica o tipo de variáveis que o ponteiro pode apontar (guardar o endereço). Tecnicamente, qualquer tipo de ponteiro pode apontar para qualquer lugar na memória. No entanto, toda a aritmética de ponteiros é feita através do tipo base do ponteiro, assim, é importante declarar o ponteiro corretamente.

8.2 OPERADORES DE PONTEIROS

&, operador unário que devolve o endereço na memória do seu operando.

m = &c; /* m recebe o endereço de c */

*****, operador unário que devolve o valor da variável localizada no endereço que o segue.

q = *m; /* q recebe o valor que está no endereço m */

Exemplo:

```
void main()
{
	float x, y;
	int *p;

	x = y = 10.098;
	p = &x;
	y = *p;
}
```

O código acima contém um erro. O ponteiro **p** aponta para um tipo base **int**, logo teremos informações perdidas, visto que o tipo **float** possui 4 bytes. O ponteiro **p** faz referência apenas aos dois primeiros bytes e não aos quatro bytes do tipo **float**.

8.3 ARITMÉTICA DE PONTEIROS

Existem duas operações: adição e subtração.

```
char *ch = 3000;
int *i = 3000;
```

ch	endereço	i
ch + 0	3000	i + 0
ch + 1	3001	
ch + 2	3002	i + 1
ch + 3	3003	
ch + 4	3004	i + 2
ch + 5	3005	

8.4 INDIREÇÃO MÚLTIPLA (PONTEIRO APONTANDO PARA PONTEIRO)

Quando falamos em indireção múltipla significa dizer que podemos ter variáveis ponteiros apontando para outras variáveis ponteiros. Variáveis ponteiros que possuem o endereço de outras variáveis ponteiros.

Exemplo:

```
1.  void main()
2.  {
3.      unsigned int a, *p1, **p2, ***p3;

4.      a = 100;
5.      p1 = &a;
6.      *p1 += *p1;
7.      p2 = &p1;
8.      **p2 = 1000;
9.      p3 = &p2;
10.     ***p3 += 1000;
11. }
```

Na linha 5, a variável ponteiro **p1** recebe o endereço da variável **a**. Na linha 6 a posição apontada pela variável ponteiro **p1** é incrementada em 100. Na linha 7, a variável 'ponteiro **p2** recebe o endereço de da variável ponteiro **p1**. Na linha 8, a posição apontada pela variável ponteiro **p2** recebe o valor inteiro 1000. Na linha 9, a variável ponteiro **p3** recebe o endereço da variável ponteiro **p2**. Na linha 10, a posição apontada pela variável ponteiro **p3** é incrementada em 1000.

8.5 PONTEIROS E STRINGS

Exemplo:

```
1.  void main()
2.  {
3.      char s[8] = "PUC-PR", *p;

4.      p = s;
5.      p[0] ^= 32;
6.      *(p + 2) ^= 040; // 40 na base 8 (octal)
7.      *(p + 4) ^= 0x20; // 20 na base 16 (hexadecimal)
8.      p[3] = '_';
9.      p++; // incrementa o endereço contido em p em 1 byte (tipo char)
10.     ++p;    // incrementa o endereço contido em p em 1 byte (tipo char)
11. }
```

Na linha 3, temos as declarações das variáveis: **s** (vetor de caracteres de tamanho igual à 8) e ***p** (variável ponteiro do tipo caracter, ou seja, que contém o endereço de uma outra variável que é do tipo caracter (**char**)).

Na linha 4, a variável ponteiro **p** recebe o endereço do primeiro byte da variável vetor **s** (**p** aponta para o índice 0 (zero) do vetor **s**).

Na linha 5, com o conteúdo da posição **p[0]**, é feito um xor (^) (operador de bits), com a constante numérica inteira 32 (base 10), atribuindo o resultado na posição **p[0]**. Na linha 6, com o conteúdo da posição ***(p + 2)**, é feito um xor (|) (operador de bits), com a constante numérica octal 040 (base 8), atribuindo o resultado na posição ***(p + 2)**. Na linha 7, com o conteúdo da posição ***(p + 4)**, é feito um xor (|) (operador de bits), com a constante hexadecimal 0x20 (base 16), atribuindo o resultado na posição ***(p + 4)**. Na linha 8, na posição **p[3]**, é atribuído a constante caractere '_'.

Nas linhas 9 e 10 são incrementados os endereços contidos na variável ponteiro **p**, agora **p** não contém mais o endereço do índice 0 do vetor **s**, mas sim o endereço do terceiro byte (índice 2), dizendo que **p** aponta para o terceiro byte de **s**.

Exemplo:

```
1.  void main()
2.  {
3.      char string[16] = "Pontifícia", *p1, *p2;

4.      p1 = p2 = string;
5.      p1++;
6.      p1++;
7.      p1++;
8.      p1 = p2;
9.  }
```

Na linha 4 tanto as variáveis ponteiro **p1** e **p2** recebem o endereço do primeiro byte da variável **string** (as variáveis ponteiro **p1** e **p2** apontam para **string**[0] ('P')). Nas linhas 5, 6 e 7, cada uma delas incrementa o endereço em um byte. Um byte porque o tipo base da variável ponteiro **p1** é **char** (caractere). Depois de executada a linha 7, a variável ponteiro **p1** aponta para o endereço de **string**[3], ou seja, o quarto byte (caractere) da variável **string**. Na linha 8 a variável ponteiro **p1** irá receber o endereço da variável ponteiro **p2**, que aponta para o primeiro byte da variável caractere **string** (**string**[0]).

8.6 PONTEIROS E VETORES

Exemplo:

```
1.  void main()
2.  {
3.      float vet[7] = {1.0,1.0,1.0,1.0,1.0,1.0,1.0}, *v;

4.      v = vet;  // v recebe o endereço do elemento 0 do vetor vet
5.      v[0] *= 1.1;
6.      v[1] -= v[0];
7.      v[2] += v[0]++ + -v[1];
8.      v[3] += 2.2 + --v[0];
9.      *(v + 4) *= sizeof(int);
10.     *(v + 5) /= 10;
11.     *(v + 6) = *(v + 0) + *(v + 1) + v[2] + v[3];
12.     v++;
13.     v[0] = 11.11;
14.     v++;
15.     *(v) = 22.22;         // *(v + 0) = 22.22;
16. }
```

8.7 PONTEIROS E MATRIZES

Exemplo:

```
1.  void main(void)
2.  {
3.      int mat[3][3] = {{0,0,0},{0,0,0},{0,0,0}}, *m;

4.      m = (int *) mat;    // m aponta para o elemento (0,0) de mat
5.      m[3*0+1] = 1;       // mat[0][1] = 1;
6.      m[3*1+1] = 1;       // mat[1][1] = 1;
7.      m[3*2+1] = 1;       // mat[2][1] = 1;
8.      m[3*0+0] += m[3*0+1] + m[3*0+2];  // mat[0][0] += m[0][1] + m[0][2];
9.      m[3*1+0] *= m[3*0+0] + m[3*0+1] + m[3*0+2];
10.     m[3*2+0] = m[3*2+1] + m[3*2+2];    // mat[2][0] = mat[2][1] + mat[2][2];
11.     m[3*1+2] = m[3*0+1] + m[3*1+1] + m[3*2+1];
12. }
```

Quando atribuímos o endereço de uma matriz para uma variável ponteiro (linha 4), a matriz deixa de ser tratada como matriz, com dois índices (linha e coluna), e passa a ser tratada como um vetor. Mesmo quando declaramos uma matriz (**int** mat[3][3]), esta está disposta na memória de uma forma linear e contígua. Na forma de ponteiro, como a matriz passa a ser tratada como vetor, temos que calcular o índice deste ponteiro em função da linha e da coluna. Por exemplo: no programa anterior, **mat**[1][2] corresponde no ponteiro **m**[3*1+2]. Isto porque o índice (1,2) equivale à: **total de colunas * número da linha + número da coluna**.

8.8 VETOR DE PONTEIROS

Podemos ter, na linguagem C, vetores de ponteiros. É um vetor onde cada posição deste vetor contém o endereço de um outra variável na memória. Cada índice do vetor armazena um endereço (aponta para uma outra variável).

Exemplo:

```
1.  void main()
2.  {
3.      char *v[4], *p = "puc-pr";
4.      char *s[3] = {"primeiro","segundo","terceiro"};

5.      v[0] = p;
6.      v[1] = s[0];
7.      v[2] = s[1];
8.      v[3] = s[2];
9.      *v[0] ^= 0x20;
10.     *(v[1]) = 'Z';
11.     *(v[1] + 4) = 'X';
12.     *v[2] = 'W';
13.     v[2][3] = 'Q';
14. }
```

Na linha 3, foi declarado um vetor de ponteiros **v** de tamanho igual à quatro (foram reservados 8 bytes na memória, supondo endereços menores que 64 Kbytes, seriam reservados 16 bytes se os endereços fossem maiores que 64 kbytes, isto dependerá do modelo de memória utilizado) e a variável ponteiro **p**. A variável ponteiro **p** aponta para o primeiro byte (caractere) da constante string "puc-pr", que se encontra armazenada na área de dados e que gasta 7 bytes (inclui o \0).

Na linha 4, aparece a declaração de um vetor de ponteiros (**s[3]**), que possui dimensão igual à três.

Na linha 5, **v[0]** recebe o endereço da variável ponteiro **p**, apontando para o primeiro byte (caractere) da constante string "puc-pr". Na linha 6, **v[1]** recebe o endereço do primeiro byte da constante **s[0]**. Na linha 7, **v[2]** recebe o endereço da constante **s[2]**. Na linha 8, **v[3]** recebe o endereço da constante **s[2]**.

Na linha 9, a variável ponteiro **v[0]** contém o endereço (aponta) para o primeiro byte da constante "puc-pr", como temos ***v[0]**, fazemos referência ao conteúdo deste endereço, onde é feito um xor (operador de bits), com a constante numérica hexadecimal 0x20 (32 em decimal ou 040 em octal).

Na linha 10, é atribuído na posição apontada por **v[1]**, a constante caractere 'Z'. Na linha 11, é atribuído na posição apontada por ***(v[1] + 4)** a constante caractere 'X'. Logo, temos ZrimXiro ao invés de primeiro.

Na linha 12, é atribuído na posição apontada por **v[2]**, a constante caractere 'W'. Na linha 12, é atribuído na posição apontada por **v[2][3]**, a constante caractere 'Q'. Logo, temos WegQndo ao invés de segundo.

No programa, acima comentado, foi mostrado as várias maneira que dispomos para fazer referência as informações apontadas pelas variáveis ponteiros existentes no programa.

Exemplo: Analise o programa abaixo e perceba o incremento e o decremento do endereço que o ponteiros possuem. Quando da depuração do programa observe na janela de watch as variáveis e também o tipo básico do ponteiro (importante).

```
1.  void main()
2.  {
3.      char s[16] = "Pontifícia", *p1, *p2;
4.      int i[6] = {1,2,3,4,5,6}, *p3, *p4;

5.      i[0] = sizeof(s);
6.      i[1] = sizeof(i);
7.      // manipulando os ponteiros para o tipo char
8.      p1 = s;   // p1 recebe o endereço s[0]
9.      p2 = &s[4];         // p2 recebe o endereço s[4]
10.     p1++;   // incrementa o endereço em 1 byte (tipo char)
11.     p1++;
12.     p2--; // decrementa o endereço em 1 byte (tipo char)
13.     p2--;
14.     // manipulando os ponteiros para o tipo int
15.     p3 = i;   // p3 recebe o endereço de i[0]
16.     p4 = &i[4];         // p4 recebe o endereço de i[4]
17.     p3++;   // incrementa o endereço em 2 bytes (tipo int)
18.     ++p3;
19.     --p4;   // decrementa o endereço em 2 bytes (tipo int)
20.     p4--;
21. }
```

8.9 EXERCÍCIOS PROPOSTOS

1. Analise o programa abaixo e informe o conteúdo das variáveis a, b e c, quando a execução do estiver na linha 10.

```
1.  void main()
2.  {
3.      int a, b, c, *p1, *p2;

4.      a = 5;
5.      b = 2;
6.      p1 = &a;
7.      p2 = &b;
8.      c = *p1 + *p2;
9.      *p1 = *p1 * *p2;
10. }
```

a = _____________ b = _____________ c = _____________

2. Analise o programa abaixo e informe o conteúdo da variável v, quando a execução do programa estiver na linha 14.

```
1.  void main()
2.  {
3.      int v[5], a, b, *p1, *p2;

4.      a = 2;
5.      b = 3;
6.      v[b - a] = 5;
7.      p1 = v;
8.      *p1 = 1;
9.      p2 = &v[b];
10.     *p2 = 13;
11.     p2++;
12.     *p2 = *(p1 + 1) + a;
13.     *(p1 + a) = a;
14. }
```

v = ____

9. VARIÁVEIS COMPOSTAS HETEROGÊNEAS

9.1 ESTRUTURAS

Em linguagem C, uma estrutura é uma coleção de variáveis referenciadas por um nome, fornecendo uma maneira conveniente de se ter informações relacionadas agrupadas. A ***definição de uma estrutura forma um modelo que pode ser usado para criar variáveis estruturas. As variáveis que compreendem a estrutura são chamadas elementos da estrutura***.

De uma forma genérica, todos os elementos de uma estrutura são logicamente relacionados. Imagine informações sobre o nome, endereço, telefone, cep, informações estas utilizadas em um programa de lista postal, seria facilmente representados através de uma estrutura.

A definição de uma estrutura para um programa em linguagem C poderia ser:

```
struct NOME
{
        [modificador] tipo identificador;
        [modificador] tipo identificador[tamanho];
        [modificador] tipo identificador[linhas][colunas];
        [modificador] tipo *identifcador;
        [modificador] tipo *identificador[tamanho];
        struct XXXX identificador;
};
```

Exemplo:

```
struct ENDEREÇO
{
        char nome[30];
        char rua[40];
        char cidade[20];
        char estado[3];
        char cep[8];
        char fone[11];
        unsigned int idade;
        unsigned int ndep;        // número de dependentes
};
```

Devemos observar que a definição termina com um ponto-e-vírgula. Isto ocorre porque a definição de estrutura, em linguagem C, é um comando. O rótulo (nome) da estrutura, identifica uma estrutura de dados particular, sendo o seu especificador de tipo.

Quando é definida uma estrutura, nesse ponto do código, **nenhuma variável foi de fato declarada**, **nenhum espaço na memória foi reservado**. Apenas a forma dos dados foi definida. Para declarar uma variável com um estrutura, devemos escrever:

struct NOME identificador;

Exemplo:

struct ENDEREÇO endereço;

O código acima declara uma variável **endereço** do tipo estrutura ENDEREÇO. Quando você define uma estrutura, está essencialmente definindo um tipo complexo de variável, não uma variável. Não existe uma variável desse tipo até que seja realmente declarada.

Exemplo:

struct ENDEREÇO *endereço;

O código acima declara uma variável ponteiro ***endereço**. Esta variável ponteiro terá o endereço de uma estrutura do tipo ENDEREÇO (aponta para um tipo de dado que é uma estrutura (**struct** ENDEREÇO)).

Exemplo:

struct ENDEREÇO endereços[10];

O código acima declara uma variável vetor de tamanho 10 (dez elementos). Cada elemento deste vetor é um dado do tipo estrutura (**struct** ENDEREÇO).

Exemplo:

struct ENDEREÇO endereços[5][10];

O código acima declara uma variável matriz de cinco linhas por 10 colunas. Cada elemento da matriz **endereços** armazena uma estrutura (**struct** ENDEREÇOS).

Exemplo:

struct ENDEREÇO *endereços[16];

O código acima declara uma variável vetor de ponteiros de tamanho igual a 16. Cada elemento do vetor de ponteiros **endereço** aponta para uma estrutura (**struct** ENDEREÇO).

O compilador da linguagem C coloca automaticamente memória suficiente para acomodar as variáveis que formam a variável estrutura.

Podemos declarar uma ou mais variáveis quando a estrutura é definida.

```
struct ENDEREÇO
{
        char nome[30];
        char rua[40];
        char cidade[20];
        char estado[3];
        char cep[8];
        char fone[11];
        unsigned int idade;
        unsigned int ndep;          // número de dependentes
} info1, info2, end1;
```

A definição da estrutura ocorre e logo após o bloco de código da estrutura, ocorreu as declarações de três variáveis do tipo da estrutura em questão. Um espaço de **sizeof**(ENDEREÇO) foi reservado na memória para cada uma das variáveis declaradas.

Para fazermos referência, manipularmos os elementos de uma estrutura, utilizaremos o operador ponto (.) e quando estivermos utilizando ponteiros, utilizaremos o operador seta (->).

9.2 EXERCÍCIOS PROPOSTOS

1. Dada a definição da estrutura:

```
struct STT
{
        int a;
        float b;
};
```

Dada a declaração: **struct** STT s;

Qual o tamanho em bytes da variável s (**sizeof**(s))? _______________

Supondo o endereço inicial da variável s igual à $3E8_{16}$. Qual o endereço de s.a e s.b? (dê o endereço em hexadecimal).

s.a = _________ s.b = _________

Dada a declaração: **struct** STT r[5];

Qual o tamanho em bytes da variável r (**sizeof**(r))? ________________

Supondo o endereço inicial da variável r igual à 1750_{8}. Qual o endereço de r[2], r[2].a e r[2].b? (dê o resultado do endereço em hexadecimal).

r[2] = __________ r[2].a = __________ r[2].b = __________

Dada a declaração: **struct** STT m[3][3];

Qual o tamanho em bytes da variável m (**sizeof**(m))? ______________

Supondo o endereço inicial da variável m igual à $7D0_{16}$. Qual o endereço de m[1][1], m[1][1].a e m[1][1].b? (dê o resultado do endereço em hexadecimal).

m[1][1] = _________ m[1][1].a = _________ m[1][1].b = _________

2. Dada as definições das estruturas:

```
struct INFO_1
{
        char s[2][3];
        int x;
};

struct INFO_2
{
        char a;
        struct INFO_1 b;
};

struct INFO_3
{
        float a;
        long int b;
        struct INFO_2 c[2];
};
```

Dada a declaração:

```
struct INFO_3 y[3];
```

Complete o quadro abaixo:

	16 bits	32 bits
sizeof(INFO_1) =	________	________
sizeof(INFO_2) =	________	________
sizeof(INFO_3) =	________	________
sizeof(y[1].c) =	________	________
sizeof(y[2].c[1].b.s[1][2]) =	________	________
sizeof(y) =	________	________

Supondo o endereço inicial da variável **y** igual à $5DC_{16}$. Quais os endereços das seguintes variáveis em arquiteturas de 16 bits e 32 bits.

____
____
____
____
____
____
____
____
____
____

	16 bits	32 bits
y[1] =	______	______
y[1].a =	______	______
y[1].b =	______	______
y[1].c[1].b.s[1][2] =	______	______
y[1].c[0].a =	______	______
y[2].c[0].b.s[0][1] =	______	______

3. Dada as definições das estruturas:

```
struct REG_1
{
        int x[2];
        char y;
};

struct REG_2
{
        char a;
        struct REG_1 b[2][2];
};

struct REG_3
{
        struct REG_1 d[3]
        struct REG_2 e[2];
};
```

Dada a declaração:

```
struct REG_3 Reg;
```

Complete o quadro abaixo:

	16 bits	32 bits
sizeof(Reg.d) =	______	______
sizeof(Reg.e[1].a) =	______	______
sizeof(Reg.e[0].b) =	______	______
sizeof(Reg.e[0].b[1][0].x[1]) =	______	______
sizeof(REG_1) =	______	______
sizeof(REG_2) =	______	______
sizeof(REG_3) =	______	______
sizeof(Reg) =	______	______

Supondo o endereço inicial da variável **Reg** igual à $7D0_{16}$. Quais os endereços das seguintes variáveis em arquiteturas de 16 bits e 32 bits.

____
____
____
____
____
____
____
____
____
____

	16 bits	32 bits
Reg.d[2].y =	____________________	____________________
Reg.e[0].b[1][0].x[1] =	____________________	____________________
Reg.e[1].a =	____________________	____________________
Reg.d[0].x =	____________________	____________________
Reg.e[1].b[1][1] =	____________________	____________________
Reg.e[1].b[1][0].y =	____________________	____________________

4. Um ponto no espaço R^2, pode ser representado por um par de números pertencentes ao conjunto dos números reais e que formam um par ordenado (x, y). Uma reta, representada no espaço R^2, pode ser definida por dois pontos. Defina os tipos de dados PONTO e RETA, este último em função de PONTO e declare um conjunto de 5 retas. Sabendo que a terceira reta passa pelos pontos (9.2,5) e (0.8,1.6), atribua estes valores ao conjunto de retas, que foi declarado.

5. Utilizando o tipo de dado PONTO, do exercício anterior, defina os seguintes tipos:

CIRCULO, definido por um ponto e o comprimento do raio.

RETA, definida por um ponto e seu coeficiente angular.

TRIANGULO_1, definido por um conjunto de três pontos.

TRIANGULO_2, definido por três pontos.

QUADRILATERO_1, definido por um conjunto de quatro pontos.

QUADRILATERO_2, definido por quatro pontos.

6. Um ponto no espaço R^3, pode ser representado por uma trinca de números pertencentes ao conjunto dos números reais e que formam uma trinca ordenada (x, y, z). Um cubo reto pode ser definido por oito pontos em R^3 e também pode ser definido por um conjunto de oito pontos. Defina os tipos PONTO, CUBO_1 e CUBO_2, estes dois últimos em função de PONTO. Declare duas variáveis do tipo CUBO_1 e CUBO_2, respectivamente e inicialize-os com os seguintes valores dos vértices: (0,0,0), (4,0,0), (4,4,0), (0,4,0), (0,0,4), (4,0,4), (4,4,4), (0,4,4).

7. Uma Reta pode ser definida por um ponto e o seu coeficiente angular. **Um plano pode ser definido por uma RETA e um PONTO, desde que este ponto não pertença a esta reta**.

Defina o tipo PLANO em função dos tipos RETA e PONTO.

Declare uma variável do tipo PLANO e atribua valores para este plano.

Declare um conjunto de cinco PLANOS e atribua valores para terceiro plano.

8. Uma data pode ser definida por três números, que representam o dia (1,...,31), mês (1,...,12, janeiro igual à 1) e ano (um número de quatro algarismos). Defina o tipo de dado DATA. **Utilize menor tipo de dado que caiba a faixa de intervalo**.

Declare duas variáveis do tipo DATA, chamadas de Inicio e Fim e atribua as datas 13/4/1992 e 29/12/1999.

9. Um endereço residencial pode ser definido através do nome da rua (35 bytes), o bairro (20 bytes), a cidade (30 bytes), a sigla do estado da federação (UF) (2 bytes), o código de endereçamento postal (Cep) (8 bytes). Defina o tipo ENDEREÇO.

10. Um telefone pode ser definido com três tipos de informações, que são: o código de discagem à distância (ddd) (3 bytes), prefixo de área (Prefixo) (4 bytes) e o número restante (Número) (4 bytes). Defina o tipo FONE.

11. O tipo de dado CLIENTE pode ser identificado pelo nome (40 bytes), número do registro geral (RG) (8 bytes), data de nascimento, endereço, fone. Defina o tipo CLIENTE, utilizando quando puder os tipos anteriores já definidos.

Declare um conjunto de 20 (vinte) clientes do tipo CLIENTE.

Supondo o endereço inicial igual à $100D_{16}$. Qual o endereço do décimo cliente. (**Dê o resultado em hexadecimal**)

12. Uma apólice de seguro de vida pode ser formada pelo número da apólice, CLIENTE, ENDEREÇO, valor da apólice, prêmio (%), nome do beneficiário (40 bytes) e data de aniversário. Defina o tipo APL_VIDA (apólice de vida).

13. Uma apólice de seguro de automóvel pode ser composta pelo número da apólice, CLIENTE, ENDEREÇO, valor da apólice, prêmio (%), depreciação, licença (10 bytes), unidade da federação (UF), modelo (15 bytes) e ano de fabricação. Defina o tipo APL_AUTO (apólice de automóvel).

14. Uma apólice de seguro de imóvel pode ser formada pelo número da apólice, CLIENTE, ENDEREÇO, valor da apólice, prêmio (%), depreciação, licença e ano de construção. Defina o tipo APL_IMOVEL (apólice de imóvel).

O exemplo abaixo mostra bem a definição de uma estrutura, a declaração da variável **info**, que é do tipo **struct** INFO e, também mostra, a manipulação de seu elementos sem ponteiros (linhas 11 a 14) e através de um ponteiro que contém o endereço da variável **info** (p aponta para a info (linha 16)).

```
struct INFO
{
	char nome[25];
	int idade;
	double altura;
};

void main()
{
	struct INFO info = {"José da Silva",30,1.82}, *p;

	// manipulando a estrutura sem ponteiros
	info.nome[0] ^= 040;
	info.nome[8] ^= 040;
	info.idade = 35;
	info.altura = 1.90;
	/* manipulando a estrutura através de um ponteiro */
	p = &info; /* p recebe o endereço de info */
	p->nome[0] ^= 0x20;
	p->nome[8] ^= 0x20;
	p->idade = 25;
	p->altura = 1.78;
}
```

No código do programa que segue, temos um ponteiro apontando para um tipo estrutura (linha 5). O ponteiro próximo contém o endereço (aponta) de uma **struct** REGISTRO. Sugere a noção de encadeamento.

```
struct REGISTRO
{
	unsigned int código;
	unsigned char nome[16];
	struct REGISTRO *proximo;
};

void main()
{
	struct REGISTRO *p;
	struct REGISTRO r1 = {1,"Cláudio",'\0'};
	struct REGISTRO r2 = {2,"Renata",'\0'};
	struct REGISTRO r3 = {3,"Marcela",'\0'};
	struct REGISTRO v[3] = {{4,"João",0},{5,"Maria",0},{6,"Renato",0}};

	// manipulando as variáveis sem ponteiros
	r1.proximo = &r2;
	r2.proximo = &r3;
	r3.proximo = &r1;
	// manipulando as variável com ponteiros
	p = &r1;
	p->proximo = &v[0];
	p->proximo->proximo = &r3;
	p->proximo->proximo->proximo = &v[1];
	p->proximo->proximo->proximo->proximo = &r2;
	p->proximo->proximo->proximo->proximo->proximo = &v[3];
}
```

9.3 UNIÕES

Na linguagem C, uma **union** é uma posição de memória que é compartilhada por duas ou mais variáveis diferente, geralmente de tipos de dados diferentes, em momentos diferentes. A definição de uma **union** é semelhante à definição de uma estrutura. Sua forma geral e:

```
union NOME
{
        [modificador] tipo identificador;
        [modificador] tipo identificador[tamanho];
        [modificador] tipo identificador[linhas][colunas];
        [modificador] tipo *identifcador;
        [modificador] tipo *identificador[tamanho];
        struct XXXX identificador;
        union YYYY identificador;
};
```

Exemplo:

```
union TIPO
{
        unsigned char c;
        unsigned int i;
};
```

A definição de uma **union** não declara quaisquer variáveis. Para declarar uma variável do tipo **union** seria:

```
union TIPO t;
```

A variável **t** é uma variável do tipo **union**. Nesta **union**, variável inteira **i** e a variável caractere **c** compartilham a mesma posição de memória. É obvio que a variável inteira **i** ocupa dois bytes na memória e a variável caractere **c** ocupa apenas um byte.

Quando uma variável **union** é declarada, o compilador cria automaticamente uma variável grande suficiente para conter o maior tipo de variável da **union**.

```
union TIPO *u;
```

A variável ponteiro **u** aponta para uma **union** (contém o endereço de uma **union**).

Neste código que é apresentado abaixo, tem-se uma união identificada por TIPO. Esta possui dois elementos que são um inteiro e um vetor de caracteres de tamanho igual à dois. O tamanho da variável **t**, que é do tipo união seria dois bytes. Observe os valores de **i**, **c[0]** e **c[1]**, quando a execução do programa estiver na linha 14.

```
union TIPO
{
        unsigned int i;
        unsigned char c[2];
};
void main(void)
{
        union TIPO t, *u;

        u = &t;
        u->i = 255;
        u->i = 256;
        u->c[0] = 0x0F;
        u->c[1] = 0x01;
}
```

O programa abaixo mostra a utilização de uma união chamada PRIMITIVA. Onde os seus elementos são quatro estruturas, que definem uma RETA, CÍRCULO, TRIÂNGULO e um QUADRADO.

```
struct PONTO_2D
{
	double x, y;
};
struct TRIÂNGULO
{
	struct PONTO_2D v[3];
};
struct QUADRADO
{
	struct PONTO_2D v[4];
};
struct CIRCULO
{
	double raio;
	struct PONTO_2D c;
};
struct RETA
{
	struct PONTO_2D p1, p2;
};
union PRIMITIVA
{
	struct CIRCULO circulo;
	struct QUADRADO quadrado;
	struct TRIÂNGULO triângulo;
	struct RETA reta;
};
void main()
{
	unsigned int t[5], i = -1;
	union PRIMITIVA a, *b;

	// preenche o vetor t com o tamanho das estruturas e da union
	t[++i] = sizeof(union PRIMITIVA);
	t[++i] = sizeof(struct CIRCULO);
	t[++i] = sizeof(struct QUADRADO);
	t[++i] = sizeof(struct TRIÂNGULO);
	t[++i] = sizeof(struct RETA);
	// manipulando os elementos da union sem ponteiro
	a.reta.p1.x = 1;
	a.reta.p1.y = 2;
	a.reta.p2.x = 1;
	a.reta.p2.y = 2;
	// manipulando os elementos da union com ponteiro
	b = &a;
	b->circulo.raio = 3.5;
	b->circulo.c.x = 3;
	b->circulo.c.y = 4;
}
```

Usar uma união ajuda na produção de códigos independentes da máquina (hardware). Como o compilador não perde o tamanho real das variáveis que perfazem a união, nenhuma dependência do hardware é produzida. Aí então, não é necessário se preocupar com o tamanho do **int**, **char**, **float** ou qualquer outra coisa que seja.

As uniões são mais utilizadas freqüentemente nas conversões de tipos quando necessárias, porque pode referenciar os dados contidos na união de maneiras diferentes.

10. ENUMERAÇÕES

Uma enumeração é uma extensão da linguagem C acrescentada pelo padrão ANSI. Uma enumeração é um conjunto de constantes inteiras que especifica todos os valores legais que uma variável desse tipo pode ter. Enumerações são comuns na vida cotidiana. Um exemplo de enumeração das moedas usadas nos Estados Unidos é: penny, nickel, dime, quarter, half-dollar, dollar.

As enumerações são definidas de forma semelhante a estruturas. A diferença é que temos a palavra **enum** usada no início de um tipo de enumeração. A forma geral de uma enumeração é:

enum NOME { *lista de enumeração* };

Para declararmos uma variável do tipo enumeração seria:

enum identificador;

Para entender bem as enumerações o ponto é que cada símbolo representa um valor inteiro e podem ser usados em qualquer lugar em que um inteiro pode ser usado. Cada símbolo é dado um valor maior em uma unidade do precedente. O valor do primeiro símbolo da enumeração é 0.

Neste exemplo, mostramos como definir, declarar e uma simples manipulação de uma enumeração.

```
enum DIAS {DOM,SEG,TER,QUA,QUI,SEX,SAB};
enum MESES {JAN, FEV, MAR, ABR = 20, MAI, JUN, JUL, AGO, SET, OUT, NOV, DEZ};
enum OPCOES {INCLUIR = -2, EXCLUIR, SAIR};

void main()
{
        enum DIAS dia[3] = {DOM,DOM,DOM};
        enum MESES mes[4] = {JAN,JAN = 10,JAN,JAN};
        enum OPCOES opcoes[3] = {SAIR,SAIR,SAIR};
        unsigned int i = -1;

        dia[++i] = DOM;
        dia[++i] = TER;
        dia[++i] = SEX;
        mes[i++ - 2] = MAR;
        mes[i++ - 2] = ABR;
        mes[i++ - 2] = MAI;
        mes[i - 2] = JUN;
        opcoes[i++ - 5] = INCLUIR;
        opcoes[i++ - 5] = EXCLUIR;
        opcoes[i - 5] = SAIR;
}
```

11. ENTRADA E SAÍDA UTILIZANDO SCANF(...) E PRINTF(...)

11.1 PRINTF(. . .)

Printf é uma rotina da biblioteca padrão que possibilita a escrita no monitor de vídeo do computador.

O protótipo desta rotina (função) encontra-se no arquivo de header STDIO.H. Esta função devolve o número de caracteres escritos ou um valor negativo se ocorrer algum erro. Seu protótipo seria:

int printf(**char** *string, [lista de argumentos, . . .]);

Para escrever qualquer string (seqüência de caracteres) no monitor de vídeo do computador, basta passarmos como parâmetro o string a ser escrito.

Exemplos da utilização do printf.

Escrever PONTIFÍCIA seria:

```
printf("PONTIFÍCIA");
```

Escrever PUC-PR 1999

```
printf("PUC-PR 1999");
```

Escrever Nome:

```
printf("Nome: ");
```

Se desejarmos escrever o conteúdo de variáveis precisamos utilizar formatos para indicar o tipo de dado a ser escrito.

Formato	Descrição
%c	Caractere
%d	Inteiro decimal com sinal
%i	Inteiro decimal com sinal
%e	Notação científica (e minúsculo)
%E	Notação científica (E maiúsculo)
%f	**Float**
%lf	**Double**
%g	Usa %e ou %f, o que for mais curto
%G	Usa %E ou %F, o que for mais curto
%o	Octal
%s	string (seqüência de caracteres)
%u	Inteiro decimal sem sinal
%x	Hexadecimal sem sinal (letras minúsculas)
%X	Hexadecimal sem sinal (letras maiúsculas)
%p	Ponteiro (endereço)
%%	escreve o símbolo %

Exemplos da utilização da função printf para escrever o conteúdo de variáveis. Para isto suponha as seguintes declarações e inicializações: **int** x = 15; **double** y = 3.1415;

Escrever o conteúdo das variáveis x e y:

```
printf("x = %d",x);
printf("y = %lf",y);
printf("x = %d - y = %lf",x,y);
```

Escrever o endereço das variáveis x e y:

```
printf("x = %p",&x);
printf("x = %p",&y);
```

11.2 SCANF(. . .)

É uma rotina da biblioteca padrão e seu protótipo está em STDIO.H. O scanf lê os dados digitados via teclado, transformando-os para o formato especificado na chamada da função scanf.

O protótipo da função é:

int scanf(**char** *formato, endereço da variável);

Especificadores de formato:

Formato	Descrição
%c	Caractere
%d	Inteiro decimal com sinal
%i	Inteiro decimal com sinal
%f	**float**
%lf	**double**
%o	Octal
%s	string (seqüência de caracteres)
%u	Inteiro decimal sem sinal
%x	Hexadecimal sem sinal (letras minúsculas)

Exemplos da utilização da função scanf para leitura de um valor entrando via teclado. Para isto suponha as seguintes declarações de variáveis: **int** x; **float** y; **char** Nome[35]; **char** Chr;

Ler as variáveis x, y, Nome, Chr:

```
scanf("%d",&x);
scanf("%f",&y);
scanf("%s",Nome); ou scanf("%s",&Nome[0]);
scanf("%c",&Chr);
```

11.3 EXERCÍCIOS PROPOSTOS

1. Faça um programa que leia um número inteiro e mostre este número em decimal, octal e hexadecimal.

2. Faça um programa que possua quatro variáveis inteiras declaradas. Mostre o endereço de cada uma das quatro variáveis declaradas no programa.

3. Faça um programa que leia quatro notas, calcule e mostre a média aritmética das notas lidas.

4. Faça um programa que leia um número real (ponto flutuante) e mostre o número lido com e sem notação científica.

5. Faça um programa que leia os catetos de um triângulo retângulo, calcule e mostre o valor da hipotenusa, com três casas decimais.

6. Construa um programa que leia os três lados de um paralelepípedo em centímetros, calcule e mostre o valor da diagonal do paralelepípedo em metros. (mostre o resultado em notação científica)

7. A partir da diagonal de um quadrado, desejamos elaborar um programa que nos informe o comprimento do lado deste quadrado. Implemente um programa que leia o valor da diagonal, calcule e mostre o valor do lado do quadrado, com uma casa decimal.

8. A conversão de graus Fahrenheit para Centígrados é obtida pela fórmula $C = \frac{5}{9}(F - 32)$. Codifique um programa que leia o valor de um temperatura em graus Fahrenheit, transforme e mostre a temperatura em graus Centígrados.

9. Para a codificação do programa que segue, utilize a seguinte definição:

```
struct PONTO
{
	double x, y, z;
};
```

Elabore um programa que leia as coordenadas de dois pontos, calcule e mostre a distância entre os dois pontos lidos, no espaço tridimensional (R^3). A distância entre dois pontos é dada pela fórmula: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$

10. Utilizando a mesma definição anterior para ponto. Implemente um programa que leia as coordenadas de dois pontos, no plano cartesiano (x, y), calcule e mostre o coeficiente angular da reta que passa pelos pontos lidos. O coeficiente angular de uma reta é dado por: $m = (y_1 - y_0) / (x_1 - x_0)$

11. Dada a definição abaixo:

```
struct CONE
{
        double Raio, Altura;
};
```

Implemente um programa que leia o raio e a altura de um cone, calcule e mostre o volume do cone.

12. Dada a estrutura abaixo:

```
struct PIRÂMIDE
{
        double Diagonal, Altura;
};
```

Construa um programa que leia a diagonal da base e a altura, de uma pirâmide de base quadrada, calcule e mostre a área da base da pirâmide e o volume.

13. Escreva um programa que calcule a quantidade de latas de tinta necessárias e o custo para pintar tanques cilíndricos de combustível, por dentro e por fora, onde são fornecidos a altura e o raio desse cilindro.

Sabemos que:

- a lata de tinta custa R$ 55,00.
- cada lata contém 5 litros.
- cada litro de tinta pintam 3 metros quadrados.

14. Analise o programa abaixo e informe o que foi escrito no monitor de vídeo do computador, quando a execução do programa estiver na linha 11.

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

2.  void main()
3.  {
4.      int n, d1, d2, d3, d4;

5.      n = 1789;
6.      d4 = n % 10;
7.      d3 = (n / 10) % 10;
8.      d2 = (n / 100) % 10;
9.      d1 = (n / 1000) % 10;
10.     printf(“%d%d%d%d”,d4,d3,d2,d1);
11. }
```

15. Dada a seguinte estrutura:

```
struct ATLETA
{
        char Nome[40];
        int Idade;
        float Altura;
};
```

Elabore um programa que leia as informações de um atleta, utilizando a estrutura acima definida e mostre as informações lidas.

16. Dadas as seguintes estruturas:

```
struct FONE
{
        int DDD, Prefixo, Número;
};
struct ALUNO
{
        int Código;
        char Nome[40];
        struct FONE Fone;
};
```

Faça um programa que leia as informações de um aluno, utilizando as estruturas acima definidas e mostre as informações lidas.

17. Utilizando a definição FONE do programa anterior, faça um programa que leia o número de um telefone e mostre o telefone no seguinte formato: (41)330-1515.

18. Dadas as seguintes definições:

```
struct DATA
{
        char Dia;
        char Mes;
        int Ano;
};
struct INTERVALO
{
        struct DATA Início;
        struct DATA Fim;
};
```

Faça um programa que leia um intervalo de horas (utilizando as estruturas definidas acima) e mostre o intervalo da seguinte forma, por exemplo:

```
Início:    10/12/1990
Fim:       11/10/1999
```

12. COMANDOS DE CONTROLE DO FLUXO DE EXECUÇÃO

O padrão ANSI divide os comandos de controle de um programa em linguagem C, nos seguintes grupos: seleção, iteração (repetição), desvio, rótulo (label), expressão e bloco.

12.1 COMANDOS DE SELEÇÃO

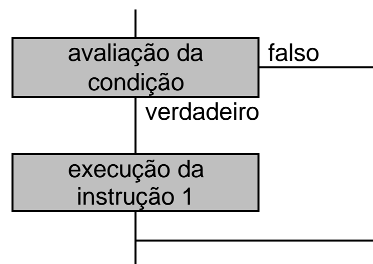
Os comandos de seleção disponíveis em linguagem C são: **if** e **switch**. Os comandos são ditos seleção porque selecionam o fluxo do programa ao longo da execução do programa (em tempo de execução).

12.1.1 COMANDO IF

Estrutura de seleção simples.

if (*condição*)
instrução 1;

Representação através de um diagrama de blocos.



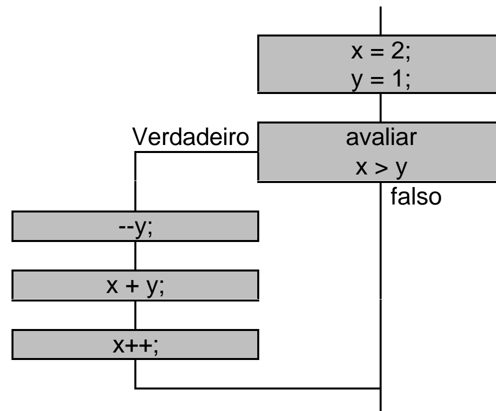
Este tipo de estrutura, que é a mais simples que existe, é analisada da seguinte maneira: se *condição*, que é uma expressão lógica, for verdadeira (1), ocorrerá a execução da instrução 1, caso contrário, se *condição* for falsa (0), não ocorrerá a execução da instrução 1.

OBS. Na linguagem C, qualquer valor diferente de zero é interpretado como verdadeiro.

Execute o programa, que segue, passo a passo e observe o conteúdo das das variáveis **x** e **y**. Feito isto, troque os valores de **x** e **y** e execute novamente o programa passo a passo. Procure observar e compreender o funcionamento do comando de seleção **if**.

```
void main()
{
	int x, y;

	x = 2;
	y = 1;
	if (x > y)
		x = x++ - --y;
}
```

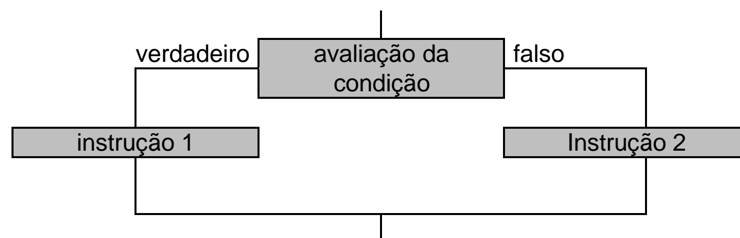
Representação do programa anterior através de um diagrama de blocos.



Estrutura de seleção composta.

```
if (condição)
        instrução 1;
else
        instrução 2;
```

Representando através de um diagrama de blocos.

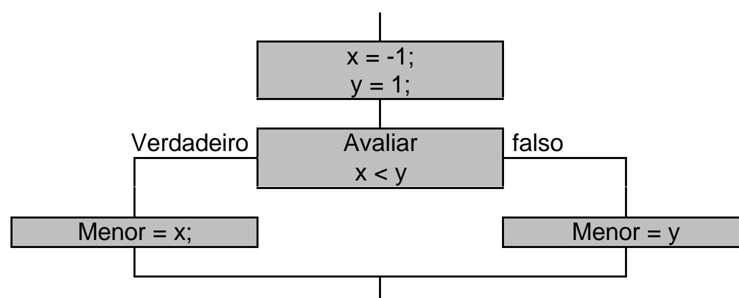


A expressão lógica *condição* é avaliada e se *condição* verdadeira (1), executa a instrução 1, caso contrário, *se condição* falsa (0), a instrução 2 é executada.

Execute o programa, que segue, passo a passo e observe o conteúdo das variáveis **x**, **y** e **Menor**. Feito isto, troque os valores de **x** e **y** e execute novamente o programa passo a passo. Procure observar e compreender o funcionamento do comando de seleção **if**.

```
void main()
{
        int x, y, Menor;

        x = -1;
        y = 1;
        if (x < y)
                Menor = x;
        else
                Menor = y;
}
```

Representação do programa anterior através de um diagrama de blocos.

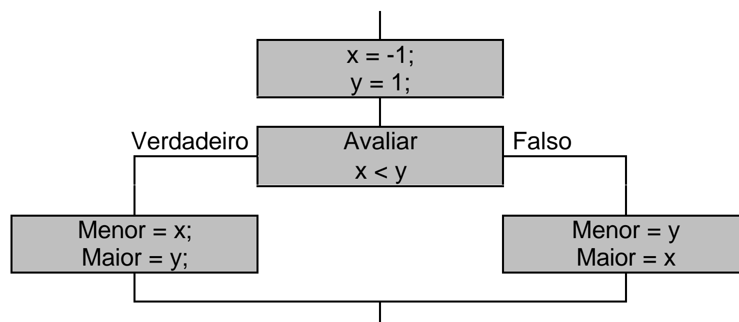


OBS. Quando tivermos mais de uma linha de código (instrução) a ser executada, dentro de um **if** ou de um **else**, somos obrigados a colocar chaves, indicando o início e final do bloco do comando. Quando tivermos apenas uma linha de código, o início e final de bloco é opcional.

Execute o programa, que segue, passo a passo e observe o conteúdo das variáveis **x**, **y**, **Maior** e **Menor**. Feito isto, troque os valores de **x** e **y** e execute novamente o programa passo a passo. Procure observar e compreender o funcionamento do comando de seleção **if**.

```
void main()
{
	int x, y, Menor, Maior;

	x = -1;
	y = 1;
	if (x < y)
	{
		Menor = x;
		Maior = y;
	}
	else
	{
		Menor = y;
		Maior = x;
	}
}
```

Representação do programa anterior através de um diagrama de blocos.



Podemos ter seqüências de código onde aparecem **if** dentro de **if** e assim por diante.

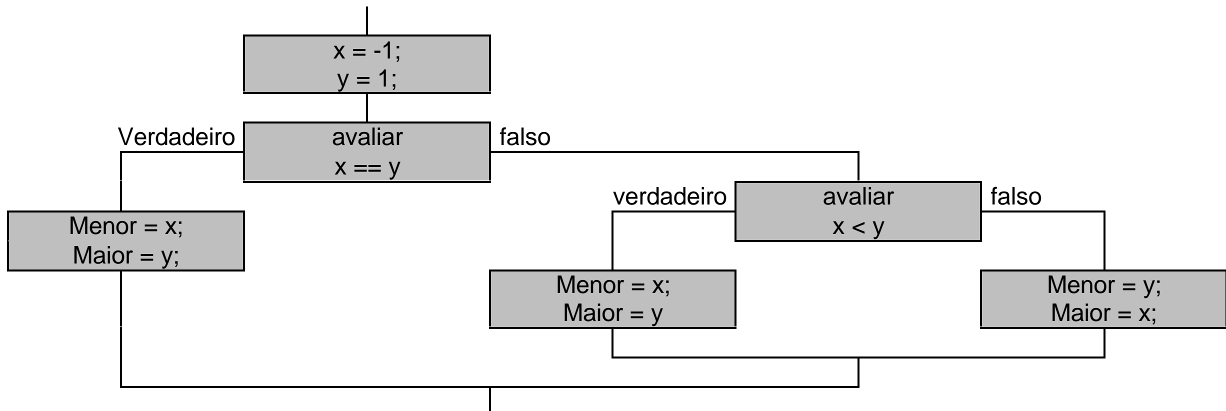
Execute o programa, que segue, passo a passo e observe o conteúdo das variáveis **x**, **y**, **Maior** e **Menor**. Feito isto, troque os valores de **x** e **y** e execute novamente o programa passo a passo. Procure observar e compreender o funcionamento do comando de seleção **if**.

```
1.  void main()
2.  {
3.          int x, y, Menor, Maior;

4.          x = -1;
5.          y = 1;
6.          if (x == y)
7.                  Menor = Maior = x;
8.          else
9.                  if (x < y)
10.                 {
11.                         Menor = x;
12.                         Maior = y;
13.                 }
14.                 else
15.                 {
16.                         Menor = y;
17.                         Maior = x;
18.                 }
19. }
```

O **else** da linha 8 possui apenas uma linha de código, que é um comando **if**, ficando opcional usar chaves para delimitar o início e final de bloco. Dependendo da experiência do programador, uma boa prática, na dúvida, colocar chaves e o programa ficaria:

```
void main()
{
        int x, y, Menor, Maior;

        x = -1;
        y = 1;
        if (x == y)
        {
                Menor = Maior = x;
        }
        else
        {
                if (x < y)
                {
                        Menor = x;
                        Maior = y;
                }
                else
                {
                        Menor = y;
                        Maior = x;
                }
        }
}
```

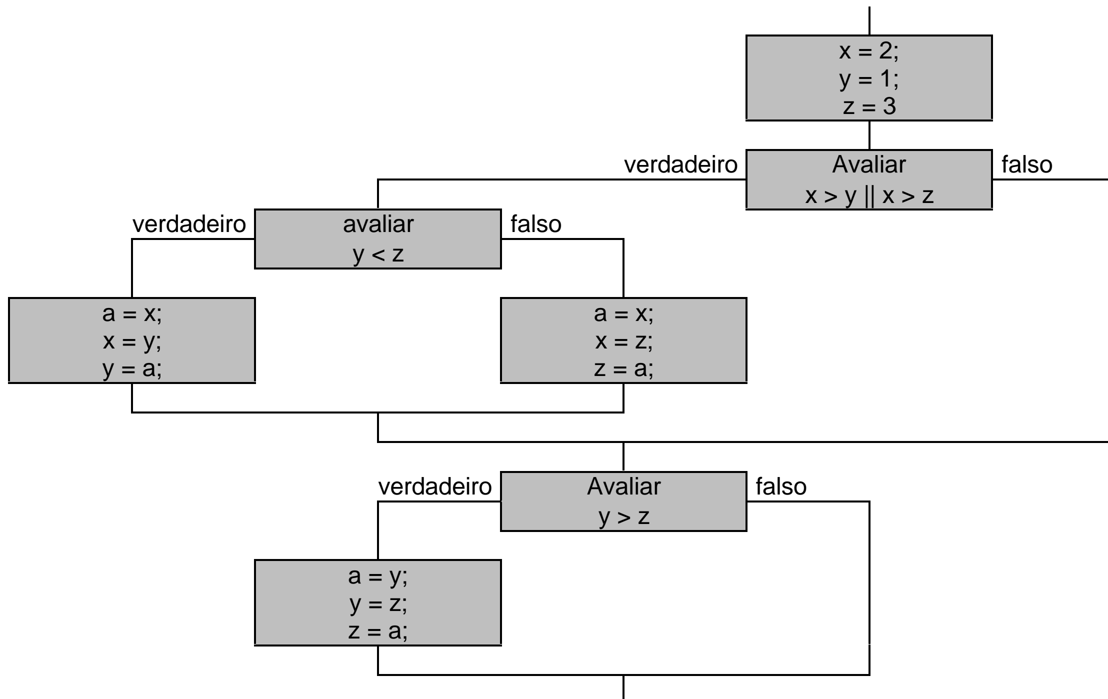
Representação do programa anterior através de um diagrama de blocos.



O programa a seguir atribui valores iniciais para as variáveis **x**, **y** e **z**. Depois verifica se estão em ordem crescente, trocando o conteúdo das variáveis, quando necessário. Observar a seqüência de código utilizado para trocar o conteúdo de duas variáveis, neste caso duas variáveis inteiras.

```
1.  void main()
2.  {
3.      int a, x, y, z;

4.      x = 2;
5.      y = 1;
6.      z = 3;
7.      if (x > y || x > z)
8.      {
9.          if (y < z)
10.         {
11.             a = x;
12.             x = y;
13.             y = a;
14.         }
15.         else
16.         {
17.             a = x;
18.             x = z;
19.             z = a;
20.         }
21.     }
22.     if (y > z)
23.     {
24.         a = y;
25.         y = z;
26.         z = a;
27.     }
28. }
```

Representação do programa anterior através de um diagrama de blocos.



O programa pode ser rescrito de uma outra maneira, tirando as chave das linhas 8 e 21, já que possuímos uma linha de código (um comando **if)**. Também poderíamos, a título de ilustração, fazer a troca entre duas variáveis inteiras utilizando o operador de bits xor (^). O programa ficaria:

```
1.  void main()
2.  {
3.      int x, y, z;

4.      x = 2;
5.      y = 1;
6.      z = 3;
7.      if (x > y || x > z)
8.              if (y < z)
9.                      x ^= y ^= x ^= y;
10.             else
11.                     x ^= z ^= x ^= z;
12.     if (y > z)
13.             y ^= z ^= y ^= z;
14. }
```

É óbvio que ao expandirmos as linhas 9, 11, 13, teríamos que colocar chaves novamente para marcar o início e fim do bloco dos comandos. Estas mesmas linhas poderiam ser rescritas da seguinte maneira, levando em contra a precedência e associatividade dos operadores, bem como, a ordem de execução que é da direita para a esquerda.

```
Linha 9 -     x = x ^ (y = y ^ (x = x ^ y));
Linha 11 -    x = x ^ (z = z ^ (x = x ^ z));
Linha 13 -    y = y ^ (z = z ^ (y = y ^ z));
```

12.1.1.1 ERROS MAIS COMUNS

Um erro bastante comum seria ao invés de colocarmos o operador de igualdade (==) colocamos o operador de atribuição (=). (linha 5)

```
1.  void main()
2.  {
3.      unsigned char ch1, ch2;

4.      ch1 = 'A';
5.      if (ch1 = 'S')
6.          ch2++;
7.      else
8.          ch2 = ch1;
9.  }
```

Na linha 5 não queremos atribuir à constante caractere 'S' para a variável caractere **ch1** e sim compararmos **ch1** com o valor da constante caractere 'S' (**if** (ch1 == 'S')). Neste caso a expressão: **ch1** = 'S', sempre será verdadeira. Para a linguagem C qualquer valor diferente de zero (0) será sempre verdadeiro.

No programa que segue temos um erro bastante comum, que seria colocarmos um ponto final na linha do comando **if**. (linha 4)

```
1.  void main(void)
2.  {
3.      char x[16] = "Puc-pr", y[16] = "Pontifícia";

4.      if (x[3] == y[3]);
5.          x[3] = y[3] = 'W';
6.  }
```

Na linha 4 aparece um ponto-e-vírgula (;) no final do comando **if** isto significa que é um **if** sem linha nenhuma a ser executada em função do resultado da expressão lógica (**x**[3] == **y**[3]). A linha 5 só deve ser executada se a expressão lógica do **if** for verdadeira (um). Com o ponto-e-vírgula isto não ocorre. A linha 5 sempre será executada. Portanto um erro que não aparece durante a compilação e a geração do programa executável, aparece sim, em tempo de execução do programa.

Outro erro bastante comum seria esquecer ou não colocar chaves (quando obrigatórias), de início e final do bloco, em uma seqüência de **if - else**.

```
void main()
{
    int x = 10, y = 9, z = 0;

    if (x > 0)
    {
        c++;
        ++y;
        if (a / c > 0)
            c += 3;
            y--;
}
```

12.1.1.2 OPERADOR INTERROGAÇÃO DOIS PONTOS (? :)

Podemos utilizar este operador para substituir comandos **if - else** na forma geral:

```
if (condição)
        instrução 1;
else
        instrução 2;
```

As expressões devem ser comandos simples, nunca um outro comando da linguagem C.

Este operador é chamado de operador interrogação dois pontos por requerer três operando. Sua forma geral e:

(*expressão 1*) **?** (*expressão 2*) **:** (*expressão 3*);

Sua avaliação é: *expressão 1* é avaliada. Se *expressão 1* verdadeira, *expressão 2* é avaliada e se torna o valor da expressão inteira. Se *expressão 1* falsa, então *expressão 3* é avaliada e se torna o valor da expressão.

Analise o programa abaixo e observe o conteúdo das variáveis, tentando compreender o funcionamento do operador interrogação dois pontos (**? :**).

```
void main(void)
{
        int x, y, Menor;

        x = -1;
        x = 1;
        Menor = (x < y) ? (x) : (y);
}
```

Analise o programa abaixo e observe o conteúdo das variáveis, tentando compreender o funcionamento do operador interrogação dois pontos (**? :**).

```
void main()
{
        int g, passo, impar;

        g = 0;
        passo = 2;
        g += (passo > 0) ? (g * 4) : (-passo);
        g -= (passo % 2) ? (impar = 1) : (3);
}
```

Analise o programa abaixo e observe o conteúdo das variáveis, tentando compreender o funcionamento do operador interrogação dois pontos (**? :**).

```
void main()
{
        int n, g, passo;

        n = 3;
        g = 0;
        passo = -2;
        n *= (!(n % 2)) ? (1) : (-1);
        g += (passo > 0) ? (passo + 3) : (-passo);
        g -= (g % 2) ? (passo++ * 3) : (--passo * g);
}
```

OBS. Não devemos utilizar indiscriminadamente o operador interrogação dois pontos (**? :**). Isto causaria um empobrecimento do código do programa quando a legibilidade, compreensão e possíveis manutenções que venham a ser necessárias.

Dado o código abaixo:

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

	a = 0;
	b = -3;
	if (a > 0)
		if (b > 0)
			c = a % b;
		else
			c = a / b;
	else
		if (b == -3)
			c = -b;
		else
			c--;
}
```

O programa anterior poderia ser rescrito utilizando o operador interrogação dois pontos (**? :**), e ficaria bem mais complexo, de difícil legibilidade e manutenção. Ficaria assim:

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

	a = 0;
	b = -3;
	c = (a > 0) ? ((b > 0) ? (a % b) : (a / b)) : ((b == -3) ? (-b) : (c--));
}
```

Uma versão para o programa anterior, para não ficaria muito ruim código quanto a legibilidade e manutenibilidade seria:

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

	a = 0;
	b = -3;
	if (a > 0)
		c = (b > 0) ? (a % b) : (a / b);
	else
		c = (b == -3) ? (-b) : (c + 1);
}
```

12.1.2 COMANDO SWITCH

A sintaxe do comando **switch** é:

```
switch (variável)
{
        case (constante 1):
                . . .;
        break;
        . . .                 . . .
        . . .                 . . .
        case (constante N):
                . . .;
        break;
        . . .                 . . .
        . . .                 . . .
        default:
                . . .;
}
```

A *variável*, analisada no comando **switch** deve ser um caractere ou um inteiro. As constantes dos vários **cases**, são constantes inteiras ou caracteres. Quando a execução do programa chegar em um comando **switch**, o conteúdo da variável é analisado e segue um dos caminhos (**case**). A palavra **default** (na falta) é opcional, pode ou não aparecer, se aparecer significa que caso nenhum dos **cases** satisfaça sempre entrará no **default**, executando as linhas de código que estiverem abaixo do **default**. Note que aparece a palavra reservada **break**, que é um desvio incondicional (sem condição). Quando é encontrado um **break**, no caso do **switch**, o fluxo da execução do programa vai para a primeira linha depois da chave que fecha o bloco de código do comando **switch**. Bloco este que é obrigatório.

O comando **switch** pode substituir uma seqüência de **if - else** que possui apenas testes de igualdades.

O programa abaixo atribui à variável **c**, o resultado da soma, da subtração, do quociente ou do resto, entre os valores **a** e **b**. Para isso é feito uma seqüência de **if - else** em função da variável opção (opção igual à 1 (um) realiza-se à soma, opção igual à 2 (dois), subtração, opção igual à 3 (três), quociente e opção igual à 4 (quatro), resto. Quando a execução do programa estiver na linha 15, o conteúdo da variável **c** terá o resultado da expressão **a** - **b**.

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

        a = 3;
        b = 4;
        opção = 2;
        if (opção == 1)     // soma
                c = a + b;
        else
                if (opção == 2)     // subtração
                        c = a - b;
                else
                        if (opção == 3)     // quociente
                                c = a / b;
                        else
                                c = a % b;          // resto
}
```

Poderíamos rescrever o código anterior utilizando o comando de seleção **switch**.

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

	a = 3;
	b = 4;
	opcao = 2;
	switch (opção)
	{
		case (1):		// soma
			c = a + b;
		break;
		case (2):		// subtração
			c = a - b;
		break;
		case (3):		// quociente
			c = a / b;
		break;
		case (4);		// resto
			c = a % b;
		break;
	}
}
```

12.1.3 EXERCÍCIOS PROPOSTOS

Um ponto em R^2 pode ser definido por um par ordenado (x, y), formando uma coordenada no plano cartesiano. Podendo ser definido como segue:

```
struct PONTO
{
	double x, y;
};
```

Uma reta pode ser representada por um ponto (definição anterior) e pelo seu coeficiente angular e pode ser definido como segue:

```
struct RETA
{
	struct PONTO p;
	double m;
};
```

Levando em conta as definições anteriores implemente os programas que seguem:

1. Faça um programa que leia duas retas, mostre as suas equações $(y = ax + b)$ e qual delas possui a maior coeficiente angular.

2. Faça um programa que leia duas retas e informe se as retas lidas são paralelas ou não. ***Duas retas, não verticais, distintas são paralelas se, e somente se, possuem o mesmo coeficiente angular.***

3. Implemente um programa que leia duas retas e informe se estas retas são perpendiculares ou não. ***Duas retas, não verticais, distintas são perpendiculares se, e somente se, o coeficiente angular de uma das retas é simétrico do inverso do coeficiente angular da outra reta.***

4. Construa um programa que leia um ponto e um reta e informe se o ponto pertence ou não a esta reta.

5. Codifique um programa que leia duas retas e se forem concorrentes, calcule e mostre ponto de intercessão. ***Duas retas, não verticais, distintas são concorrente se não forem paralelas.***

Tanto um cone como um cilindro pode ser definido pelo seu raio e altura, como segue:

```
struct CONE
{
        double Raio;
        double Altura;
};
struct CILINDRO
{
        double Raio;
        double Altura;
};
```

Levando em conta as definições anteriores, implemente os programas que seguem:

6. Implemente um programa que leia dois cubos e informe qual dos dois possui o maior volume.

7. Codifique um programa que leia um cone, calcule e mostre a altura que um cilindro deve ter de modo que possua o mesmo volume do cone. O cilindro e o cone possuem a mesma base.

8. Desejamos calcular, a partir do sexo e da altura, o peso ideal de uma pessoa. Para isto devemos saber que existem duas fórmulas, que são:

para homens: $peso_ideal = (72{,}7 * altura) - 58$;

para homens: $peso_ideal = (62{,}1 * altura) - 44{,}7$;

Para a implementação deste programa utilize a definição abaixo:

```
struct PESSOA
{
        char Nome[40];
        char Sexo;
        int Idade;
        double Altura;
};
```

Elabore um programa que leia as informações de uma pessoa e informe se esta pessoa está obesa ou não. Para que uma pessoa seja considerada obesa, a diferença entre o seu peso e o peso ideal deve ser superior à 40 quilogramas.

9. Codifique um programa que leia o nome e o peso de um boxeador e informe à categoria a qual o boxeador pertence, seguindo a tabela abaixo:

Categoria	Peso (Kg)
Palha	menor que 50 Kg
Pluma	50 - 59,999
Leve	60 - 75,999
Pesado	76 - 87,999
Super Pesado	maior ou igual à 88

Utilize na implementação a estrutura abaixo:

```
struct BOXEADOR
{
        char Nome[40];
        float Peso;
};
```

10. Faça um programa que escolha a moeda a ser lida (1 - libra, 2 - franco, 3 - dólar, 4 - marco ou 5 - real), leia um montante, transforme (conforme tabela) e mostre o valor equivalente nas outras moedas. Utilize para isto uma estrutura chamada MOEDA, que possui os seguinte elementos: Libra, Franco, Dólar, Marco e Real.

Moeda	Valor (R$)
1 dólar americano	1,12
1 libra esterlina	0,96
1 franco francês	1,37
1 marco alemão	1,15

11. Faça um programa que leia três valores A, B e C, verifique se eles podem ser os comprimentos dos lados de um triângulo e, se forem, verificar se compõem um triângulo equilátero, isósceles ou escaleno. Informar também se não compuserem nenhum triângulo.

Lembramos que:

- comprimento de cada lado de um triângulo é menor do que a soma dos comprimentos dos outros dois lados.
- Chama-se triângulo equilátero ao triângulo que tem os comprimentos dos três lados iguais.
- Chama-se triângulo isósceles ao triângulo que tem os comprimentos de dois lados iguais. Portanto, todo triângulo equilátero é também isósceles.
- Chama-se triângulo escaleno ao triângulo que tem os comprimentos de seus três lados diferentes.

12. Faça um programa que seja capaz de concluir qual dentre os animais seguintes foi escolhido, através de perguntas e respostas. Animais possíveis: onça, boi, porco, homem, mico leão, morcego, golfinho, avestruz, pingüim, pato, condor, jabuti, jacaré, sucuri.

exemplo:
É mamífero? S
É quadrúpede? S
É carnívoro? N
É herbívoro? S
Então o animal escolhido foi o boi.

- mamífero	- quadrúpede	- carnívoro	- onça
		- herbívoro	- boi
		- onívoro	- porco
	- bípede	- onívoro	- homem
		- frutífero	- mico leão
	- voadores	- morcego	
	- aquáticos	- golfinho	
- aves	- não voadoras	- tropical	- avestruz
		- polar	- pingüim
	- nadadoras	- pato	
	- de rapina	- condor	
- répteis	- com casco	- jabuti	
	- carnívoro	- jacaré	
	- sem patas	- sucuri	

13. Faça um programa que leia um número real e informe se este número possui ou não casas decimais (parte fracionária).

14. Elabore um programa que leia um número real e mostre o número arredondado. Os critérios são: 2,2 vai para 2,0; 2.8 vai para 3,0; 2,5 vai para 3,0.

12.2 COMANDOS DE REPETIÇÃO

A linguagem C possui três comandos estruturados de repetição, que são: **while**, **do - while** e o **for**. Os comandos de repetição são responsáveis por permitirem que um bloco de código seja repetido por um número finito de vezes. Quem controla o número de vezes (iterações) que o comando faz são as expressões lógicas.

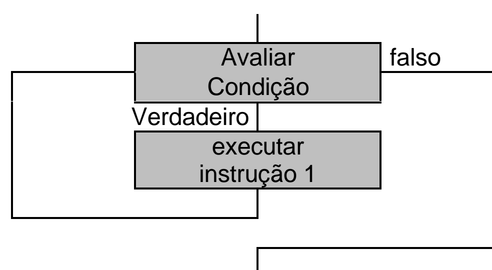
12.2.1 COMANDO DE REPETIÇÃO WHILE

O comando de repetição **while** caracteriza-se por fazer o teste no início. Por isso muitas vezes é dito comando de repetição com teste no início. Isto implica que nem sempre o código dentro do bloco do comando será executado. Isto elimina a necessidade de efetuar um teste condicional antes do comando.

A forma geral de um comando **while** seria:

```
while (condição)
        instrução 1;
```

Representação através de um diagrama de blocos.



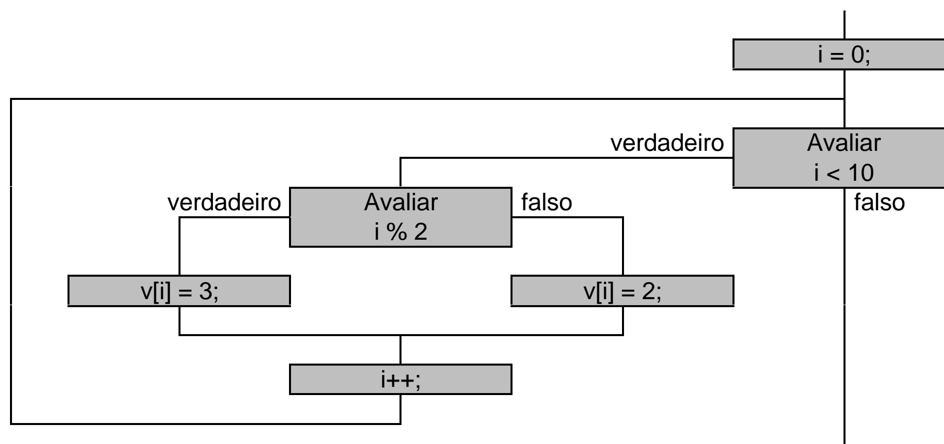
Onde *condição* é uma expressão lógica. Enquanto o resultado da expressão lógica for verdadeiro (na linguagem C verdadeiro é qualquer valor diferente de 0 (zero)) a instrução 1 é executada. Caso contrário, a instrução 1 não será executada.

OBS. Vale lembrar que se tivermos mais que uma linha de código (instrução) a ser executada dentro de uma comando **while**, é obrigatório a utilização de chaves ({,}), para marcar o início e final do bloco de código do comando **while**.

O programa a seguir preenche um vetor de 10 posições da seguinte maneira. As posições de índice ímpar é preenchido com o valor três e as posições de índice par com o valor dois.

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

        i = 0;
        while (i < 10)
        {
                if (i % 2)
                        v[i] = 3;
                else
                        v[i] = 2;
                i++;
        }
}
```

Representação do programa anterior através de um diagrama de blocos.



O programa anterior poderia ser rescrito utilizando o operador interrogação dois pontos (**? :**). A seqüência de código ficaria:

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

	i = 0;
	while (i < 10)
		v[i++] = (i % 2) ? (3) : (2);
}
```

Exemplo: O programa abaixo calcula a média aritmética de um vetor de números reais e também extrai o maior e o menor número do vetor. O vetor é inicializado na declaração.

```
void main()
{
	unsigned int i;
	double vetor[7] = {1.1, 2.3, 2.0, 4.6, 5.0, 3.0, 5.6};
	double média, menor, maior, acumulador;

	menor = 99;
	maior = acumulador = i = 0;
	while (i < sizeof(vetor) / sizeof(double))
	{
		acumulador = acumulador + v[i];
		if (v[i] < menor)
			menor = v[i];
		else
			if (v[i] > maior)
				maior = v[i];
		i++;
	}
	média = acumulador / (sizeof(vetor) / sizeof(double));
}
```

12.2.2 COMANDO DE REPETIÇÃO DO - WHILE

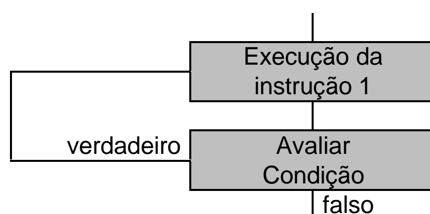
O comando de repetição **do - while** possui o teste no final do comando. Isto garante que uma iteração sempre será feita.

A forma geral de um comando **do - while** seria:

do
instrução 1;
while (*condição*);

Onde *condição* é uma expressão lógica. Enquanto o resultado da expressão lógica for verdadeiro (**na linguagem C verdadeiro é qualquer valor diferente de 0 (zero)**) a instrução 1 é executada. Caso contrário, a instrução 1 não será executada.

Representação através de um diagrama de blocos.



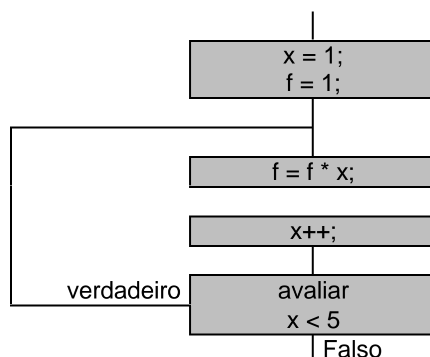
OBS. Se tivermos mais que uma linha de código a ser executada dentro do comando **do - while**, é obrigatório a colocação de chaves para indicar o início e final do bloco do comando **do - while**.

OBS. É bom salientar que o comando **do – while** sempre acaba com um ponto-e-vírgula (;) na linha da declaração do **while**.

O programa abaixo calcula o fatorial do número 4.

```
1.  void main()
2.  {
3.      int x, f;

4.      x = f = 1;
5.      do
6.      {
7.          f *= x;
8.          x++;
9.      } while (x < 5);
10. }
```

Representação do programa anterior através de um diagrama de blocos.



O programa anterior poderia ser rescrito da seguinte maneira:

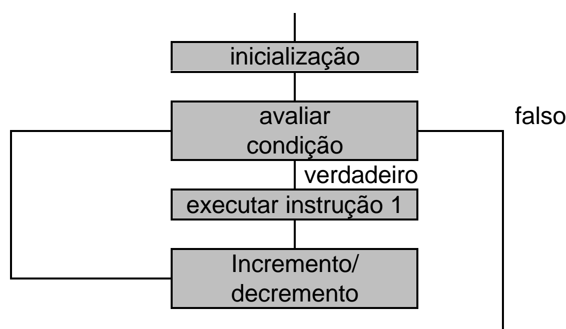
```
void main()
{
        int x, f;

        x = f = 1;
        do
                f *= x++;
        while (x < 5);
}
```

12.2.3 COMANDO DE REPETIÇÃO FOR

O comando de repetição **for** é também chamado de comando de repetição com variável de controle. Na sua declaração existem três expressões que são: *inicialização*, *condição* e *incremento/decremento*. O interessante deste comando é que o controle todo está na sua declaração. A *inicialização* e *incremento/decremento* são expressões aritméticas, enquanto *condição* é uma expressão lógica, que controlará o número de repetições.

A forma geral de um comando **for** seria:

for (*inicialização*; *condição*; *incremento/decremento*)
instrução 1;



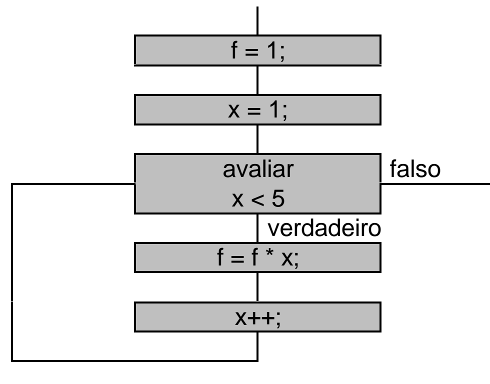
OBS. Se tivermos mais que uma linha de código, é obrigatório a utilização de chaves ({,}), para marcar o início e final do bloco de códigos pertencentes ao comando **for**.

O programa abaixo calcula o fatorial do número 4.

```
1.  void main()
2.  {
3.          int x, f;

4.          f = 1;
5.          for (x = 1; x < 5; x++)
6.                  f *= x;
7.  }
```

Quando a execução do programa estiver na linha 5, temos um comando **for**. Será realizada a inicialização da variável **x** com o valor 1, a condição é testada e se **for** verdadeira (para a linguagem C, qualquer valor diferente de zero é verdadeiro), executa a linha 6. Depois de executada uma interação, é incrementado o conteúdo da variável **x**, neste caso em um.

Representação do programa anterior através de um diagrama de blocos.



Os programas que seguem, geram a matriz descrita abaixo, de maneira diferentes, utilizando os vários comandos de repetições existentes na linguagem C.

SZZZZ
YSZZZ
YYSZZ
YYYSZ
YYYYS

Com o comando **for**:

```
void main()
{
	char matriz[5][5], i, j;

	for (i = 0; i < 5; i++)
		for (j = 0; j < 5; j++)
			if (i == j)
				matriz[i][j] = ‘W’;
			else
				if (i < j)
					matriz[i][j] = ‘Z’;
				else
					matriz[i][j] = ‘Y’;
}
```

Com o comando **while**:

```
void main()
{
	char matriz[5][5], i, j;

	i = 0;
	while (i < 5)
	{
		j = 0;
		while (j < 5)
		{
			if (i == j)
				matriz[i][j] = 'W';
			else
				if (i < j)
					matriz[i][j] = 'Z';
				else
					matriz[i][j] = 'Y';
			j++;
		}
		i++;
	}
}
```

Com o comando **do - while**

```
void main()
{
	char matriz[5][5], i, j;

	i = 0;
	do
	{
		j = 0;
		do
		{
			if (i == j)
				matriz[i][j] = 'W';
			else
				if (i < j)
					matriz[i][j] = 'Z';
				else
					matriz[i][j] = 'Y';
			j++;
		} while (j < 5);
		i++;
	} while (i < 5);
}
```

Observe o código abaixo e analise o incremento das variáveis **i** e **j**.

```
void main()
{
	char matriz[5][5], i, j;

	i = 0;
	do
	{
		j = 0;
		do
		{
			if (i == j)
				matriz[i][j] = 'W';
			else
				if (i < j)
					matriz[i][j] = 'Z';
				else
					matriz[i][j] = 'Y';
		} while (++j < 5);
	} while (++i < 5);
}
```

12.3 COMANDOS DE DESVIOS INCONDICIONAIS

12.3.1 COMANDO BREAK

O comando **break** quando utilizado dentro de uma estrutura de repetição, provoca um desvio do fluxo do programa para fora do comando de repetição. Evitando o teste condicional normal do laço.

O código a seguir mostra uma utilização simples do comando **break**.

```
void main()
{
	int t, c = 0;

	for (t = 0; t < 100; t++)
	{
		c++;
		if (t == 5)
			break;
	}
}
```

Quando a execução do programa acima estiver na linha 9, um salto para a linha 11 será dado, ou seja, a execução do programa irá para esta linha.

12.3.2 COMANDO CONTINUE

O comando **continue** quando utilizado dentro de uma estrutura de repetição, provoca um desvio do fluxo de modo que ocorra uma nova iteração, pulando qualquer código intermediário.

O código a seguir mostra uma utilização simples do comando **continue**.

```
void main()
{
        char string[36] = "Ca sa", *s;
        int contador;

        s = string;
        for (contador = 0; *s; s++)
        {
                s++;
                if (*s != ' ')
                        continue;
                contador++;
        }
}
```

12.4 EXERCÍCIOS PROPOSTOS

12.4.1 SÉRIES

1. Fazer um programa que calcule e escreva o valor de *S*.

$$S = \frac{1}{1} + \frac{3}{2} + \frac{5}{3} + \frac{7}{4} + \ldots + \frac{99}{50}$$

2. Fazer um programa que calcule e escreva a seguinte soma:

$$\frac{2^1}{50} + \frac{2^2}{49} + \frac{2^3}{48} + \ldots + \frac{2^{50}}{1}$$

3. Fazer um programa para calcular e escrever a seguinte soma:

$$S = \frac{37x38}{1} + \frac{36x37}{2} + \frac{35x36}{3} + \ldots + \frac{1x2}{37}$$

4. Fazer um programa que calcule e escreva o valor de *S* onde:

$$S = \frac{1}{1} - \frac{2}{4} + \frac{3}{9} - \frac{4}{16} + \frac{5}{25} - \frac{6}{36} \ldots \frac{-10}{100}$$

5. Fazer um programa que calcule e escreva a soma dos 50 primeiros termos da seguinte série:

$$\frac{1000}{1} - \frac{997}{2} + \frac{994}{3} - \frac{991}{4} + \ldots$$

6. Fazer um programa que calcule e escreva a soma dos 30 primeiros termos da série:

$$\frac{480}{10} - \frac{475}{11} + \frac{470}{12} - \frac{465}{13} + \ldots$$

7. Fazer um programa para gerar e escrever uma tabela com os valores do seno de um ângulo *A* em radianos, utilizando a série de Mac-Laurin truncada, apresentada a seguir:

$$\text{sen } A = A - \frac{A^3}{6} + \frac{A^5}{120} - \frac{A^7}{5040}$$

Condições: os valores dos ângulos *A* devem variar de 0,0 a 6,3, inclusive, de 0,1 em 0,1.

8. O valor aproximado de p pode ser calculado usando-se a série:

$$S = \frac{1}{1^3} - \frac{1}{3^3} + \frac{1}{5^3} - \frac{1}{7^3} + \frac{1}{9^3} - \ldots$$

sendo [ ]. Fazer um programa para calcular e escrever o valor de p com 51 termos.

9. Fazer um programa que calcule e escreva o valor de *S*, através do somatório que segue:

$$S = \frac{1}{225} - \frac{2}{196} + \frac{4}{169} - \frac{8}{144} + \ldots + \frac{16384}{1}$$

10. Fazer um programa que calcule e escreva a soma dos 20 primeiros termos da série:

$$\frac{100}{0!} + \frac{99}{1!} + \frac{98}{2!} + \frac{97}{3!} + \ldots$$

11. Fazer um programa que calcule e escreva a soma dos 50 primeiros termos da série:

$$\frac{1!}{1} - \frac{2!}{3} + \frac{3!}{7} - \frac{4!}{15} + \frac{5!}{31} - \ldots$$

12.4.1.1 RESPOSTA DOS EXERCÍCIOS SOBRE SÉRIES

Exercício 1 - Soma = 95.50079466167059650000

Exercício 2 - Soma = 1560828692041339.75000000000000000000

Exercício 3 - Soma = 4080.75078370370875000000

Exercício 4 - Soma = 0.64563492063492078400

Exercício 5 - Soma = 685.29690205764632100000

Exercício 6 - Soma = 21.11071558638006710000

Exercício 7
Seno 0.0 = 0.00000000000000000000
Seno 0.1 = 0.09983341664682540700
Seno 0.2 = 0.19866933079365081700
Seno 0.3 = 0.29552020660714289900
Seno 0.4 = 0.38941834158730159700
Seno 0.5 = 0.47942553323412701000
Seno 0.6 = 0.56464244571428567900
Seno 0.7 = 0.64421757652777778000
Seno 0.8 = 0.71735572317460316600
Seno 0.9 = 0.78332584982142849900
Seno 1.0 = 0.84146825396825386500
Seno 1.1 = 0.89120093311507930800
Seno 1.2 = 0.93202505142857139400

Seno 1.3 = 0.96352940640873019400
Seno 1.4 = 0.98539379555555561800
Seno 1.5 = 0.99739118303571427900
Seno 1.6 = 0.99938856634920636400
Seno 1.7 = 0.99134644299603169600
Seno 1.8 = 0.97331677714285702600
Seno 1.9 = 0.94543936628968239200
Seno 2.0 = 0.90793650793650770800
Seno 2.1 = 0.86110586624999974100
Seno 2.2 = 0.80531143873015831300
Seno 2.3 = 0.74097252287698367500
Seno 2.4 = 0.66855058285714219500
Seno 2.5 = 0.58853391617063421900
Seno 2.6 = 0.50142002031745946900
Seno 2.7 = 0.40769555946428470700
Seno 2.8 = 0.30781383111110993700
Seno 2.9 = 0.20216963275793514700
Seno 3.0 = 0.09107142857142705440
Seno 3.1 = -0.02528928394841438790
Seno 3.2 = -0.14687150730158918300
Seno 3.3 = -0.27382127732143063400
Seno 3.4 = -0.40651129650793882400
Seno 3.5 = -0.54558376736111369200
Seno 3.6 = -0.69199652571428849100
Seno 3.7 = -0.84707257406746339100
Seno 3.8 = -1.01255311492063838000
Seno 3.9 = -1.19065418410714674000
Seno 4.0 = -1.38412698412698765000
Seno 4.1 = -1.59632201748016200000
Seno 4.2 = -1.83125712000000274000
Seno 4.3 = -2.09368949418650985000
Seno 4.4 = -2.38919184253968364000
Seno 4.5 = -2.72423270089285730000
Seno 4.6 = -3.10626107174603039000
Seno 4.7 = -3.54379545759920322000
Seno 4.8 = -4.04651739428570867000
Seno 4.9 = -4.62536958430554712000
Seno 5.0 = -5.29265873015871779000
Seno 5.1 = -6.06216316767855368000
Seno 5.2 = -6.94924539936505603000
Seno 5.3 = -7.97096962771822248000
Seno 5.4 = -9.14622438857138853000
Seno 5.5 = -10.49585038442455160000
Seno 5.6 = -12.04277361777771380000
Seno 5.7 = -13.81214392446420550000
Seno 5.8 = -15.83147900698402740000
Seno 5.9 = -18.13081406783717850000
Seno 6.0 = -20.74285714285699460000
Seno 6.1 = -23.70315023454347170000
Seno 6.2 = -27.05023634539660900000
Seno 6.3 = -30.82583251124974310000
Seno 6.4 = -35.07500893460287020000

Exercício 8 - Soma = 3.14158881890944786000

Exercício 9 - Soma = 14693.63808699286530000000

Exercício 10 - Soma = 269.10990101744556600000

Exercício 11 - Soma = -2.59749559877651011000000000000000000000000e+49

12.4.2 VETORES

1. Faça um programa que leia um vetor de inteiros de oito elementos e mostre este vetor na mesma linha, com um espaço entre os elementos.

2. Faça um programa que leia um vetor de reais de tamanho igual à dez e mostre o valor do somatório dos elementos do vetor.

3. Faça um programa que leia um vetor de inteiros de tamanho igual à quinze e informe o menor elemento do vetor.

4. Implemente um programa que leia um vetor de inteiros de tamanho igual à doze e informe o maior e o menor valor existente neste vetor.

5. Implemente um programa que leia um vetor de inteiros de tamanho igual à quinze, calcule e mostre a amplitude dos elementos. ***A amplitude é a diferença entre o maior e o menor elemento do vetor***.

6. Faça um programa que leia um vetor qualquer de dez posições, calcule e mostre a média aritmética dos elementos deste vetor.

7. Faça um programa que leia dois vetores de caracteres, de tamanhos iguais à cinco e oito respectivamente. Gere um terceiro vetor que possua as informações dos dois vetores lidos respetivamente (um na seqüência do outro).

10. Construa um programa que leia um vetor de quinze elementos quaisquer. Calcule a variância e o desvio padrão dos dados deste vetor. Onde:

$$Variancia = \sum_{i=0}^{n-1} (v_i - media)^2 \text{ e } desvio = \sqrt{\mathrm{var}\,iancia}\,.$$

11. Faça um programa que leia uma seqüência de caracteres (string) e conte e mostre o número de vogais desta seqüência de caracteres.

12. Implemente um programa que leia um vetor, de caracteres, de tamanho igual à dez, mostre o vetor lido, inverta e mostre o vetor invertido. Para realizar a inversão, tente não utilizar outro vetor, faça a inversão no mesmo vetor.

13. Faça um programa que leia um valor (base dez) inteiro, maior ou igual à zero contido no intervalo 1 a 255 inclusive, tranforme e mostre este valor na base dois. Faça este programa de duas maneira, utilizando um vetor e sem vetor. Faça também uma consistência do número lido, informando que o número não é válido (fora do intervalo pedido). O programa acaba quando **for** lido o valor 0 (zero).

12.4.3 MATRIZES

1. Faça um programa que leia uma matriz (5x5) de caracteres e mostre a matriz no formato de matriz que estamos acostumados.

2. Construa um programa que leia duas matrizes de ordem três e mostre a soma destas duas matrizes.

3. Implemente um programa que leia duas matrizes de ordem três, calcule e mostre o resultado da multiplicação das matrizes.

4. Faça um programa que lê uma matriz (4x2), lê um escalar e mostra o resultado da multiplicação da matriz lida pelo escalar.

5. Faça um programa que gere e mostre a seguinte matriz:

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2

6. Faça um programa que gere e mostre a seguinte matriz:

2	2	2	2	2	1	1	1	1	1
2	2	2	2	2	1	1	1	1	1
2	2	2	2	2	1	1	1	1	1
2	2	2	2	2	1	1	1	1	1
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2

7. Faça um programa que gere e mostre a seguinte matriz:

2	2	2	1	1	1	1	1	1	1
2	2	2	1	1	1	1	1	1	1
2	2	2	1	1	1	1	1	1	1
2	2	2	1	1	1	1	1	1	1
1	1	1	1	1	1	2	2	2	2
1	1	1	1	1	1	2	2	2	2
1	1	1	1	1	1	2	2	2	2
1	1	1	1	1	1	2	2	2	2
1	1	1	1	1	1	2	2	2	2
1	1	1	1	1	1	2	2	2	2

8. Faça um programa que gere e mostre a seguinte matriz:

A	A	A	A	A	A	A	A	A	A
A	A	A	A	A	A	A	A	A	A
A	A	B	B	B	B	B	B	A	A
A	A	B	B	B	B	B	B	A	A
A	A	B	B	B	B	B	B	A	A
A	A	B	B	B	B	B	B	A	A
A	A	B	B	B	B	B	B	A	A
A	A	B	B	B	B	B	B	A	A
A	A	A	A	A	A	A	A	A	A
A	A	A	A	A	A	A	A	A	A

9. Faça um programa que gere e mostre a seguinte matriz:

```
0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1
```

10. Faça um programa que gere e mostre a seguinte matriz:

```
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
```

11. Faça um programa que gere e mostre a seguinte matriz:

```
0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 1 1 1 1
```

12. Faça um programa que gere e mostre a seguinte matriz:

```
A A A A A A A A A A
A A A A A A A A A A
A A A A A A A A A A
A A A A A A A A A A
A A A A A A A A A A
A A A A A A A A A A
A A A A A A A A A A
B B B B B B B B B B
B B B B B B B B B B
B B B B B B B B B B
```

13. Faça um programa que gere e mostre a seguinte matriz:

```
A A A A A A A A A A A A A A A A
A A A A A A A A A A A A A A A A
A A B B B B B B A A B B B B A A
A A B B B B B B A A B B B B A A
A A B B B B B B A A B B B B A A
A A B B B B B B A A B B B B A A
A A A A A A A A A A B B B B A A
A A A A A A A A A A B B B B A A
A A A A A A A A A A B B B B A A
A A B B B B B B A A B B B B A A
A A B B B B B B A A B B B B A A
A A B B B B B B A A B B B B A A
A A B B B B B B A A B B B B A A
A A A A A A A A A A B B B B A A
A A A A A A A A A A A A A A A A
A A A A A A A A A A A A A A A A
```

14. Construa um programa que leia uma matriz (4x3), calcule e mostre o somatório dos valores contidos linha por linha. Por exemplo:

1	2	1	→	4
1,1	0,1	2	→	2,2
2	0,2	0,1	→	2,3
5	1	2	→	8

15. Implemente um programa que leia uma matriz (4x3), calcule e mostre o somatório dos valores contidos coluna por coluna. Por exemplo:

1	2	1
1,1	0,1	2
2	0,2	0,1
5	1	2
↓	↓	↓
9,1	3,3	5,1

12.4.4 EXERCÍCIOS GERAIS SOBRE REPETIÇÕES

1. Faça um programa que leia um valor do tipo **float** (32 bits) e mostre como este **float** está armazenado na memória do computador, mostrando o conteúdo e a seqüência dos bytes que representam o **float** lido. Exemplo: o número 2513.125, seria mostrado:

```
8D06:0FFF - 45
8D06:0FFE - 1D
8D06:0FFD - 12
8D06:0FFC - 00
```

2. Elabore um programa que leia um valor inteiro e mostre a tabuada deste número, da seguinte maneira. Suponha que o número lido seja o 5, o programa mostraria:

5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50

3. Faça um programa que leia o valor N, um número inteiro positivo e calcule o valor do seguinte somatório:

$$S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots + \frac{1}{N}$$

4. O número 3025 possui a seguinte característica:

$$\begin{cases} 30 + 25 = 55 \\ 55 * 55 = 3025 \end{cases}$$

Construa um programa que pesquise e mostre todos os números de quatro algarismos que apresentam tal característica.

5. Faça um programa que pesquise, no intervalo de 0 a 200, os números perfeitos. ***Um número perfeito é aquele cuja soma de seus divisores, exceto ele próprio, é igual ao número***. Exemplo: 6 é perfeito porque 1 + 2 + 3 = 6.

6. Faça um programa que mostre os números primos no intervalo de 100 a 200. ***Um número é primo se e somente se ele for divisível por 1 ou ele mesmo***.

7. Faça um programa que leia um número inteiro e positivo e mostre este número invertido. ***Sem utilizar vetor***. Exemplo: foi lido 3456 o programa mostrará 6543.

8. A quadrado de um número inteiro e positivo é igual à soma dos n primeiros termos ímpares. Faça um programa que leia um número inteiro e positivo, calcule o quadrado deste número e mostre-o em seguida.
Exemplo: $3^2 = 1 + 3 + 5 = 9$

$$n^2 \sum_{i=0}^{n-1} (2 * i + 1)$$

13. FUNÇÕES

As funções são uma das características mais importantes da linguagem C. As funções são, nada mais nada menos, que, blocos de códigos, identificados por um nome, onde toda a atividade do programa ocorre. O código de uma função é um bloco discreto. O código de uma função é privativo à função e nenhum outro comando pode ter acesso em uma outra função. Exceto através de uma chamada de função.

A forma geral de uma função é:

[modificador] tipo nome_da_função(*[modificador]* tipo identificador 1, . . .)
[modificador] tipo *nome_da_função(*[modificador]* tipo identificador 1, . . .)

Onde, o nome_da_função seria um identificador para a função, que seguem as mesmas regras do identificador de uma variável.

O nome de uma função deve ter uma seqüência significativa, por exemplo: Média, RetornaMédia, MaiorValor LeNotas, LeNota, LeNumero, LeReal, LeNome, RetTurmaAluno, RetornaTurma, RetornaMediaBimestral, MédiaBimestral, CalculaImposto, MontaMatriz, CalculaVetor, RetNomeUsuario, MontraQuadro, Círculo, Moldura, RetornaCaractere, RetornaSexo, Simpson, Gauss, MulMatriz, SomaMatriz, Inversa, MatrizInversa, RetornaElemento, GeraMatriz, GeraVetor, CopiaString, StrTamanho, StrConcatena e etc.

O nome dever ser compatível com o que a função faz. Como as funções da biblioteca padrão da linguagem C, são todas escritas com letras minúsculas, é de bom grado **utilizar nomes para as funções que comecem com letras maiúsculas, como vimos no parágrafo anterior**.

Tudo que está antes do nome da função é o especificador de valor que o comando **return** da função devolve, podendo ser qualquer tipo válido. Se nenhum tipo é especificado, o **default** (o que compilador assume) é que a função devolve um valor inteiro.

A função também pode não retornar nada. Aparecendo o tipo **void**, especificando um conjunto vazio de valores. O **void** é usado como tipo retornado por funções que não retornam nada.

As informações que estão após o nome da função, entre parênteses, são os parâmetros, que se trata de uma lista de declarações de variáveis. Sendo uma lista, estas variáveis são separadas por vírgulas, quando existirem mais que uma declaração.

Uma função pode não ter parâmetro algum, neste caso a lista de parâmetros é vazia. Apenas abrem-se e fecham-se os parênteses ou escrevemos **void** dentro dos parênteses.

Os parâmetros (variáveis declaradas) e as variáveis declaradas dentro do bloco de código da função, são ditas variáveis locais, locais ao bloco de código, (locais na função). As variáveis locais não podem manter os seus valores entre as chamadas de funções. A única exceção são para as variáveis estáticas (**static**), fazendo com que o compilador trate à variável com se ela fosse uma variável global, para fins de armazenamento, mas ainda limita sua atuação para dentro da função.

Uma variável é dita global, quando **for** declarada fora da(s) função(es). Esta variável está armazenada na área de dados e sempre é iniciada com o valor zero (0, NULL, '\0'). Uma variável global pode ser utilizada dentro de qualquer uma das funções, daí o termo: **a variável é visível em todo o programa**.

Na linguagem C, todas as funções estão no mesmo nível de escopo. Obedecem as mesmas regras de escopo que governam se um bloco de código conhece ou tem acesso a outro bloco de código ou dados. Não é possível definir uma função dentro de uma outra função. Isto mostra que a linguagem C não é uma linguagem estruturada em blocos.

O programa abaixo mostra a utilização de funções em um programa. O programa possui três funções (main, Quadrado e Cubo). Observe também que podemos chamar uma função dentro de outra função.

```
1.  double Cubo(double);
2.  double Quadrado(double);

3.  double c, q;

4.  void main()
5.  {
6.      double v;

7.      v = 3;
8.      q = Quadrado(v);
9.      c = Cubo(3) + Quadrado(Cubo(v - 1) - 2);
10. }

11. double Cubo(double x)
12. {
13.     double r;

14.     r = Quadrado(x) * x;
15.     return (r);
16. }

17. double Quadrado(double x)
18. {
19.     double r;

20.     r = x * x;
21.     return (r);
22. }
```

Nas linhas 1 e 2, aparecem os protótipos das funções Quadrado e Cubo. O protótipo (prototype) de uma função avisa ao compilador quais os tipos e quantos argumentos (parâmetros) a função receberá e o tipo de dado que retornará. **Quando a função main estiver antes da seqüência das funções, é obrigado colocar os protótipos das funções no programa.**

Observe a declaração das variáveis **c** e **q**, que estão fora das funções, logo são ditas **variáveis globais.**

A variável **v**, declarada na linha 8 é uma variável local ao bloco de código da função main. Esta variável só é visível (manipulada) dentro do bloco de código desta função.

As variáveis **x** e **r**, declaradas nas linhas 11 e 13, são variáveis locais ao bloco de código da função Cubo, sendo visíveis apenas neste bloco.

As variáveis declaradas nas linhas 17 e 19 (**x** e **r**), são variáveis locais ao bloco de código da função Quadrado. Apesar de possuírem o mesmo nome que as variáveis locais à função Cubo, são variáveis diferentes, posições diferentes na memória.

Nas linhas 8, 9 e 14 ocorrem chamadas de funções. Quando ocorre uma chamada de função, o fluxo do programa é deslocado para o bloco de código da função chamada.

Na linha 9, primeiro é chamada a função Cubo(**v** - 1), em seguida, a função Quadrado. O parâmetro passado para a função Quadrado seria o valor obtido do retorno da função Cubo(**v** - 1), chamada anteriormente, menos o valor 2 (dois). Por último teríamos chamada da função Cubo passando o parâmetro 3 (três). Assim o valor retornado pela função Quadrado mais o resultado retornado pela função Cubo(3), é atribuído para a variável **c**.

Quando chamamos uma função ocorrem os seguinte eventos:

1. Empilha os parâmetros (da direita para a esquerda).
2. Empilha o endereço de retorno.
3. Chama à função.
4. Se houver variáveis locais reservar espaço na pilha.
5. Executa às instruções.
6. Coloca o valor de retorno em um registrador.
7. Desempilha o endereço de retorno e para lá retorna.

Nas linhas 15 e 21, aparece o comando **return**, que é um comando de desvio incondicional (não possui condição). Este comando tem dois importantes usos: **primeiro** - ele força uma saída da função; **segundo** - o **return** é usado para retornar um valor da função.

13.1 PASSAGEM DE PARÂMETROS POR VALOR

A passagem de parâmetros por valor copia o valor de um argumento no parâmetro formal da sub-rotina. Neste caso os valores são passados para a função e precisam ser guardados em variáveis parâmetros declaradas. O(s) valor(es) podem são utilizados dentro do bloco de código da função e depois do retorno da função são destruídos e os valores são perdidos.

Usaremos o exemplo anterior para mostrar a passagem de parâmetros por valor. Rescrevemos o programa apenas de uma forma diferente.

```
1.  double Cubo(double);
2.  double Quadrado(double);

3.  double c, q;

4.  void main()
5.  {
6.        double v;

7.        v = 3;
8.        q = Quadrado(v);
9.        c = Cubo(3) + Quadrado(Cubo(v - 1) - 2);
10. }

11. double Cubo(double x)
12. {
13.       return (Quadrado(x) * x);
14. }

15. double Quadrado(double x)
16. {
17.       return (x * x);
18. }
```

OBS. Uma cópia do valor do argumento que é passado para a função. O que ocorre dento da função não tem efeito algum sobre a variável usada na chamada.

13.2 PASSAGEM DE PARÂMETROS POR REFERÊNCIA

Nesta maneira de passar um parâmetro, o endereço de um argumento é copiado no parâmetro. Dentro da função (sub-rotina), o endereço é usado para acessar o argumento real utilizado na chamada. Isso significa que alterações feitas no parâmetro afetam a variável usada para chamar a rotina.

O programa que segue, lê três valores inteiros (**x**, **y** e **z)**, para depois colocá-los em ordem crescente, trocando os valores de duas variáveis quando necessário.

```
1.  void main()
2.  {
3.      int a, x, y, z;

4.      printf("x = ");
5.      scanf("%d",&x);
6.      printf("y = ");
7.      scanf("%d",&y);
8.      printf("z = ");
9.      scanf("%d",&z);
10.     if (x > y || x > z)
11.     {
12.         if (y < z)
13.         {
14.             a = x;
15.             x = y;
16.             y = a;
17.         }
18.         else
19.         {
20.             a = x;
21.             x = z;
22.             z = a;
23.         }
24.     }
25.     if (y > z)
26.     {
27.         a = y;
28.         y = z;
29.         z = a;
30.     }
31.     printf("%d - %d - %d",x,y,z);
32. }
```

Podemos perceber seqüências de códigos que podem ser reunidas em uma função. Seriam as seqüências em que funcionalmente realizam a mesma coisa, como: trocarmos os valores de duas variáveis e lermos um número inteiro.

O programa ficaria:

```
1.  int LerInteiro(char *);
2.  void Trocar(int *, int *);

3.  void main()
4.  {
5.      int a, x, y, z;

6.      x = LerInteiro("x = ");
7.      y = LerInteiro("y = ");
8.      z = LerInteiro("z = ");
9.      if (x > y || x > z)
10.         if (y < z)
11.             Trocar(&x,&y);
12.         else
13.             Trocar(&x,&z);
14.     if (y > z)
15.         Trocar(&y,&z);
16.     printf("%d - %d - %d",x,y,z);
17. }
```

```
18. int LerInteiro(char *s)
19. {
20.       int n;

21.       printf(s);
22.       scanf("%d",&n);
23.       return (n);
24. }

25. void Trocar(int *a, int *b)
26. {
27.       int aux;

28.       aux = *a;
29.       *a = *b;
30.       *b = aux;
31. }
```

A função Trocar é uma função que não retorna nada por isso no seu protótipo aparece, a palavra reservada, **void** para informar que a função na realidade retorna um conjunto vazio de valores.

Quando é feita a chamada da função Trocar, está sendo passado, como parâmetros, o endereço das variáveis e não o seu valor. A declaração dos parâmetros é preciso declará-los como ponteiros para o mesmo tipo do endereço passado. Dentro da função Trocar, ocorrerá à troca dos dois valores. Existindo a necessidade de uma variável auxiliar (**aux**) e será manipulado o conteúdo do endereço passado, isto será feito através do operador unário *****.

Como a função Trocar manipula o conteúdo de endereços, quando do seu término, o conteúdo (valor) armazenados nos endereços passados estarão modificados.

A função LerInteiro recebe como parâmetro o título do campo, de modo a orientar qual variável está sendo entrada (digitada).

O programa anterior poderia ser rescrito, tendo na função main as chamadas de funções LerNumeros e OrdenarTresValores, o programa ficaria:

```
1.  void OrdenarTresValores(int *, int *, int *);
2.  void LerNumeros(int *, int *, int *);
3.  void Trocar(int *, int *);

4.  void main()
5.  {
6.        int a, b, c;

7.        LerNumeros(&a,&b,&c);
8.        OrdenarTresValores(&a,&b,&c);
9.        printf("%d - %d - %d",a,b,c);
10. }

11. void OrdenarTresValores(int *x, int *y, int *z)
12. {
13.       if (*x > *y || *x > *z)
14.             if (*y < *z)
15.                   Trocar(x,y);
16.             else
17.                   Trocar(x,z);
18.             if (y > z)
19.                   Trocar(y,z);
20. }
```

```
21. void Trocar(int *a, int *b)
22. {
23.       int aux;

24.       aux = *a;
25.       *a = *b;
26.       *b = aux;
27. }

28. void LerNumeros(int *x, int *y, int *z)
29. {
30.       printf("x = ");
31.       scanf("%d",x);
32.       printf("y = ");
33.       scanf("%d",y);
34.       printf("z = ");
35.       scanf("%d",z);
36. }
```

A função LerNumeros passa os endereços das variáveis onde os valores lidos do teclados serão colocados. Note que a passagem é por referência.

No exemplo anterior, foi feito a função OrdenarTresValores, que recebe os três valores por referência e coloca os três valores em ordem.

OBS. Os dois últimos códigos escritos, fazem a mesma coisa só que escritos de maneiras diferentes. Isto é importante por que vai conseguindo um refinamento do programa, deixando-o bastante funcional. Algumas destas funções podem bem ser utilizadas em outros programass, reutilizando o código sempre que possível.

13.3 CHAMANDO FUNÇÕES, PASSANDO E RETORNANDO ESTRUTURAS

Dado o programa: o código que segue lê o nome e o número do telefone, mostrando-os em seguida, utilizando para isto uma estrutura.

```
1.  struct INFO
2.  {
3.        char Nome[20];
4.        char Fone[12];
5.  };

6.  void main()
7.  {
8.        struct INFO r;

9.        printf("Nome: ");
10.       scanf("%s",&r.Nome);
11.       printf("Fone: ");
12.       scanf("%s",&r.Fone);
13.       printf("Nome: %s - Fone: %s\n",r.Nome,r.Fone);
14. }
```

O programa anterior pode ser escrito de uma outra maneira. Vamos escrevê-lo utilizando duas funções, que seriam: LerInfo e MostrarInfo.

```
1.  struct INFO
2.  {
3.      char Nome[20];
4.      char Fone[12];
5.  };

6.  void LerInfo(struct INFO *);
7.  void MostrarInfo(struct INFO);

8.  void main()
9.  {
10.     struct INFO r;

11.     LerInfo(&r);
12.     MostrarInfo(r);
13. }

14. void LerInfo(struct INFO *a)
15. {
16.     printf("Nome: ");
17.     scanf("%s",&a->Nome);
18.     printf("Fone: ");
19.     scanf("%s",&a->Fone);
20. }

21. void MostrarInfo(struct INFO a)
22. {
23.     printf("Nome: %s - Fone: %s\n",a.Nome,a.Fone);
24. }
```

A função LerInfo recebe como parâmetro um dado do tipo **struct** INFO. A informações lidas via teclado, deverão ser colocadas em um lugar na memória que é a variável a. A passagem do parâmetro é por referência, por que desejamos que o conteúdo exista depois do encerramento da função.

Poderíamos escrever a função LerInfo de uma outra maneira. Note que funcionalmente, que a função LerInfo, faz a mesma coisa.

```
1.  struct INFO
2.  {
3.      char Nome[20];
4.      char Fone[12];
5.  };

6.  struct INFO LerInfo();
7.  void MostrarInfo(struct INFO);

8.  void main()
9.  {
10.     struct INFO r;

11.     r = LerInfo();
12.     MostrarInfo(r);
13. }

14. void LerInfo()
15. {
16.     struct INFO a;

17.     printf("Nome: ");
18.     scanf("%s",&a.Nome);
19.     printf("Fone: ");
20.     scanf("%s",&a.Fone);
21.     return (a);
22. }

23. void MostrarInfo(struct INFO a)
24. {
25.     printf("Nome: %s - Fone: %s\n",a.Nome,a.Fone);
26. }
```

Note que a função LerInfo retorna um tipo de dado, **struct** INFO, por isso é obrigatório o uso de um **return** e da atribuição deste retorno a uma posição de memória.

13.4 CHAMANDO FUNÇÕES, PASSANDO UM VETOR COMO PARÂMETRO

Podemos passar como parâmetro, para uma função, um vetor. Este parâmetro sempre será declarado, na função, como ponteiro do tipo do vetor que foi passado.

OBS. A passagem de parâmetro de vetores é **sempre** por referência (sempre é passado o endereço do primeiro elemento ([0]) e recebido na função como um ponteiro do tipo do vetor passado.

O programa que segue calcula a média aritmética do conteúdo de um vetor de inteiros, que representam as idades de doze pessoas. O programa irá ler o vetor de idades, calcular a média e mostrar a média das idades.

```
1.  void main()
2.  {
3.      int Idades[12], i;
4.      double Media = 0;

5.      for (i = 0; i < 12; i++)
6.      {
7.              printf("Idades[%d] = ",i);
8.              scanf("%lf",&Idades[i]);
9.      }
10.     for (i = 0; i < 12; i++)
11.             Media += Idades[i] / 12.0;
12.     printf("Media = %.2lf\n",Media);
13. }
```

O programa anterior poderia ser escrito utilizando uma função que lê o conteúdo do vetor de idades e uma outra função que calcula a média das idades do vetor. O programa ficaria assim:

```
1.  void LerIdades(int *);
2.  double CalcularMedia(int *);

3.  void main()
4.  {
5.      int Idades[12];
6.      double Media;

7.      LerIdades(Idades);
8.      Media = CalcularMedia(Idades);
9.      printf("Media = %.2lf\n",Media);
10. }

11. void LerIdades(int *v)
12. {
13.     int i;

14.     for (i = 0; i < 12; i++)
15.     {
16.             printf("v[%d] = ",i);
17.             scanf("%lf",&v[i]);
18.     }
19. }

20. double CalcularMedia(int *v)
21. {
22.     int i;
23.     double m = 0;

24.     for (i = 0; i < 12; i++)
25.             m += v[i] / 12.0;    // m += *(v + i) / 12.0;
26.     return (m);
27. }
```

O programa que segue lê um vetor de pontos no plano cartesiano e mostra estes pontos. Inicialmente o programa sem funções:

```
1.  struct PONTO
2.  {
3.      double x, y;
4.  };

5.  void main()
6.  {
7.      struct PONTO p[5];
8.      int i;

9.      for (i = 0; i < 5; i++)
10.     {
11.             printf("Ponto %d:\n",i);
12.             printf("x = ");
13.             scanf("%lf",&p[i].x);
14.             printf("y = ");
15.             scanf("%lf",&p[i].y);
16.     }
17.     for (i = 0; i < 5; i++)
18.             printf("Ponto %d : (%lf,%lf)\n",i,p[i].x,p[i].y);
19. }
```

Rescrevendo o programa com duas novas funções (LerPontos e MostrarPontos) teríamos:

```
1.  struct PONTO
2.  {
3.      double x, y;
4.  };

5.  void LerPontos(struct PONTO *);
6.  void MostrarPontos(struct PONTO *);

7.  void main()
8.  {
9.      struct PONTO p[5];

10.     LerPontos(p);
11.     MostrarPontos(p);
12. }

13. void LerPontos(struct PONTO *v)
14. {
15.     int i;

16.     for (i = 0; i < 5; i++)
17.     {
18.             printf("Ponto %d:\n",i);
19.             printf("x = ");
20.             scanf("%lf",&v[i].x);
21.             printf("y = ");
22.             scanf("%lf",&v[i].y);
23.     }
24. }
```

```
25. void MostrarPontos(struct PONTO *v)
26. {
27.         int i;

28.         for (i = 0; i < 5; i++)
29.                 printf("Ponto %d : (%lf,%lf)\n",i,v[i].x,v[i].y);
30. }
```

13.5 CHAMANDO FUNÇÕES, PASSANDO UMA MATRIZ COMO PARÂMETRO

Podemos passar como parâmetro, para uma função, uma matriz. Este parâmetro sempre será declarado, na função, como ponteiro do tipo da matriz que foi passada.

OBS. A passagem de parâmetro de matrizes é **sempre** por referência (sempre é passado o endereço do primeiro elemento ([0][0]) e recebido na função como um ponteiro do tipo da matriz passada.

O programa que segue lê uma matriz (3x2) e mostra o conteúdo da matriz, no formato de matriz, que estamos acostumados.

```
1.  void main()
2.  {
3.          int i, j;
4.          double Matriz[3][2];

5.          clrscr();
6.          for (i = 0; i < 3; i++)
7.                  for (j = 0; j < 2; j++)
8.                  {
9.                          printf("m[%d][%d] = ",i,j);
10.                         scanf("%lf",&m[i][j]);
11.                 }
12.         for (i = 0; i < 3; i++)
13.         {
14.                 for (j = 0; j < 2; j++)
15.                         printf("%.1lf",m[i][j])
16.                 printf("\n");
17.         }
18.         getch();  // apenas para parar a tela
19. }
```

Podemos rescrever o programa utilizando funções. Criaremos duas funções: LerMatriz e MostrarMatriz. A função LerMatriz vai ler o conteúdo da matriz e colocar as informações na memória, a partir do endereço recebido como parâmetro. O novo programa ficaria:

OBS. Quando é passado uma matriz como parâmetro para uma função, dentro da função não podemos utilizar os índices linha e coluna, como é de costume. Isto porque temos dentro da função um ponteiro para um endereço e a matriz está linearmente na memória, a partir deste endereço.

```
1.  void LerMatriz(double *);
2.  void MostrarMatriz(double *);

3.  void main()
4.  {
5.      int i, j;
6.      double Matriz[3][2];

7.      clrscr();
8.      LerMatriz((double *) Matriz,3,2);
9.      MostrarMatriz((double *) Matriz,3,2);
10.     getch();  // apenas para parar a tela
11. }

12. void LerMatriz(double *m, int l, int c)
13. {
14.     int i, j;

15.     for (i = 0; i < l; i++)
16.         for (j = 0; j < c; j++)
17.         {
18.             printf("m[%d][%d] = ",i,j);
19.             scanf("%lf",&m[c * i + j]);
20.         }
21. }

22. void MostrarMatriz(double *m, int l, int c)
23. {
24.     int i, j;

25.     for (i = 0; i < l; i++)
26.     {
27.         for (j = 0; j < c; j++)
28.             printf("%.1lf",m[c * i + j])
29.         printf("\n");
30.     }
31. }
```

14. ALOCAÇÃO DINÂMICA DE MEMÓRIA

Na linguagem C, podemos reservar blocos de bytes de memória dinamicamente, durante a execução do programa. O espaço reservado estaria no heap.

Para reservarmos memória no heap, nós temos a função malloc (memory allocation), da biblioteca padrão. O seu protótipo é: **void** *malloc(**size_t**), que recebe como parâmetro o número de bytes a ser reservados no heap e retorna um ponteiro para o primeiro byte do bloco reservado. Se não existe memória suficiente no heap, a função malloc retorna NULL (0,'\0'). O protótipo pode ser encontrado em: <stdlib.h>, <alloc.h>.

É conveniente verificarmos sempre se conseguimos reservar memória, utilizando um **if** e abortarmos o programa sempre que não conseguimos reservar espaço na memória.

Sempre que um programa reservou memória no heap, no mesmo programa, deve ser liberada a memória, através da função free, que possui o seguinte protótipo: **void** free(**void** *), que é encontro em <stdlib.h>, <alloc.h>. A função free recebe como parâmetro o endereço do primeiro byte de um bloco.

O programa abaixo lê uma seqüência de caracteres (string) e mostra em seguida. O espaço reservado para a seqüência de caracteres, que é lida através da alocação dinâmica de memória.

```
1.  void main()
2.  {
3.        char *Str;

4.        Str = (char *) malloc(35);   // reserva memória no heap
5.        if (!Str)
6.        {         // não conseguiu reservar memória no heap
7.                  puts("Faltou memória\a\n . . .");
8.                  exit(0);
9.        }
10.       printf("Nome: ");
11.       gets(Str);
12.       printf("Nome lido: %s\n",Str);
13.       free(Str);            // libera memória
14.       getch();
15. }
```

O programa a seguir, lê um vetor de cinco inteiros e mostra o conteúdo deste vetor. Utilizando alocação dinâmica de memória.

```
1.  void main()
2.  {
3.        int i, *v;

4.        v = (int *) malloc(10);       // malloc(sizeof(int) * 5));
5.        if (!v)
6.        {         // não conseguiu reservar memória no heap
7.                  puts("Faltou memória\a\n . . .");
8.                  exit(0);
9.        }
10.       for (i = 0; i < 5; i++)
11.       {
12.                 printf("v[%d] = ");
13.                 scanf("%d",&v[i]);
14.       }
15.       for (i = 0; i < 5; i++)
16.                 printf("%d ",v[i]);   // printf("%d ",*(v + i));
17.       free(v);   // libera memória
18.       getch();
19. }
```

15. EXERCÍCIOS PROPOSTOS

15.1 STRINGS

OBS.

Deve ser tomado o maior cuidado possível quanto à manipulação de uma seqüência de caracteres na linguagem C. Como existe uma flexibilidade grande de programação, fica a cargo do programador, que deve tomar cuidado com a invasão do espaço de memória de uma outra memória. O programador deve ter uma visão ampla do que está sendo feito e pensar em tudo que **for** necessário.

Quando manipulamos seqüências de caracteres, com a linguagem C, devemos pensar no barra invertida zero ('\0', 0 ou NULL). Existem situações que é necessário o '\0', outras não. O caractere barra invertida zero, na linguagem C, tem a função de finalizar (indicar o fim) de uma seqüência de caracteres.

Quando o retorno de uma função, após a manipulação de uma seqüência de caracteres, que ocorre por referência deve-se tomar cuidado para que esta seqüência de caracteres tenha espaço reservado dentro do endereço passado para função.

OBS. Procure utilizar com e ou sem alocação dinâmica de memória e observe os códigos gerados.

1. Fazer um programa, que leia uma seqüência de caracteres (string) pelo teclado, através da função **gets**, da biblioteca padrão. O programa deve mostrar o tamanho da seqüência de caracteres lida. O tamanho do string (seqüência da caracteres) deve ser calculada através de uma função chamada **StrTam**, que recebe como parâmetros um string e retorna o tamanho do string.

2. Fazer um programa, que leia uma seqüência de caracteres (string) pelo teclado, através da função **gets**, da biblioteca padrão. O programa deve copiar o conteúdo do string lido para uma outra variável (outra posição de memória). Está cópia deve ser feita através de uma função chamada **StrCopia**, que recebe dois parâmetros, dois strings. O protótipo da função seria: **void StrCopia(char *d, char *s)**, que copia o conteúdo da variável do ponteiro **s** para a variável **d**. **Obs.** ***Note o tamanho reservado na memória para as variáveis em questão, principalmente depois da cópia realizada*.**

3. Fazer um programa, que leia duas seqüências de caracteres (strings) pelo teclado, através da função **gets**, da biblioteca padrão. O programa deve mostrar os dois strings concatenados (o segundo string lido deve ser colocado, na seqüência, logo após o primeiro). A concatenação deve ser feita através da função **StrCat**, que recebe dois parâmetros, dois strings. O protótipo da função seria: **void StrCat(char *d, char *s)**, que põe o conteúdo da variável ponteiro **s**, na seqüência da variável ponteiro **d**, gerando a partir do ponteiro **d** o string concatenado. **Obs.** ***Note o tamanho reservado na memória para a variável ponteiro d aponta, deve ser suficiente para receber o conteúdo das variáveis ponteiros d e s*.**

4. Fazer um programa, que leia uma seqüência de caracteres (string) pelo teclado, através da função **gets**, da biblioteca padrão. O programa deve mostrar o número de caracteres, letras, vogais e consoantes do string lido. Para isto devemos construir as seguintes funções:

- **ContarVogais**, que recebe um string como parâmetro e retorna o número de vogais do string.
- **ContarConsoantes**, que recebe um string como parâmetro e retorna o número de consoantes do string.
- **ContarLetras**, que recebe um string como parâmetro e retorna o número de letras do string.

5. No mesmo programa anterior adicione a seguinte definição:

```
struct AVAL
{
        int Vogais;
        int Consoantes;
        int Letras;
        int Caracteres;
};
```

Construa três novas funções que farão a mesma coisa, só que de maneiras diferentes. As funções são:

struct AVAL PesquisarString(char *s), que recebe a seqüência de caracteres a ser analisada e a função retorna uma **struct** AVAL, preenchida corretamente.

void PesquisarString(char *s, struct AVAL *), que recebe a seqüência de caracteres a ser analisada e, por referência, retorna uma **struct** AVAL, preenchida corretamente.

int PesquisarString(char *s, int *v, int *c, int *l), que recebe a seqüência de caracteres a ser analisada e, por referência, retorna o número de vogais, consoantes e de letras. A função retorna o número de caracteres do string passado para a análise.

6. A função **gets** possui o incômodo, de não controlar o tamanho do string a ser lido. Faça um programa que leia e mostre um string. Para lermos o string, devemos construir a função **LerString**, que recebe como parâmetros um string e o tamanho máximo do string a ser lido. O protótipo seria: **void LerString(char *s, int t)**. A função **LerString** retorna o string lido na variável **s**, passada por referência. A função é encerrada quando:

A tecla **RETURN** (código **ASCII 13**) **for** pressionada, colocando um '\0' no final do string.

A tecla **ESC** (código **ASCII 27**) **for** pressionada, colocando um '\0' no primeiro byte do string, retornando assim um string nulo).

O número de caracteres a serem digitados não pode exceder a **t - 1**, onde **t** foi passado como parâmetro.

7. Rescreva a função **LerString**, de modo que possua o seguinte protótipo: **void LerString(char *s, int t, int tipo)**, onde o tipo é o tipo do caractere a ser lido, que fará parte do string e que pode ser: 0 - **ALFABÉTICO** (só caracteres alfabéticos), 1 - **ALFANUMÉRICO** (só caracteres alfabéticos e ou numéricos), 2 - **NUM_INTEIRO** (só caracteres numéricos), 3 - **NUM_REAL** (só caracteres numéricos e o caractere ponto) e 4 - **SEM_TIPO** (qualquer caractere ASCII). A função não deixa ser lido um caractere que não faça parte do tipo de caractere. **Sugestão:** ***utilize #defines para os tipos mencionados*.**

8. Faça um programa, que leia uma seqüência de caracteres numéricos (utilize a função **LerString** do exercício anterior), transforme e mostre o string lido em um inteiro. Para fazer esta transformação, escreva uma função que possua o seguinte protótipo: **int RetornarInteiro(char *)**, que recebe como parâmetro uma seqüência de caracteres numéricos (string numérico) e que retorna um inteiro.

9. Faça um programa, que leia uma seqüência de caracteres numéricos mais o caractere ponto (utilize a função **LerString** dos exercícios anteriores), transforme e mostre o string lido em um real. Para fazer esta transformação, escreva uma função que possua o seguinte protótipo:**double RetornarReal(char *)**, que recebe como parâmetro uma seqüência de caracteres numéricos (string numérico) com ou sem o caractere ponto e que retorna um inteiro.

10. Faça um programa que lê dois números: um inteiro e um real. A leitura destes dois números devem ser feitas através de duas funções, que são: **Obs.** ***Os valores devem ser lidos caractere a caractere usando a função getch(), da biblioteca padrão, colocando-os em um vetor de caracteres finalizado sempre com o caracteres '\0'. Utilize as funções dos exercícios 7, 8 e 9, respectivamente.***

int LerInteiro(int n), que recebe como parâmetro o número máximo de algarismos que possui o número a ser lido por esta função.

double LerReal(int n), que recebe como parâmetro o número máximo de algarismos que possui o número a ser lido por esta função.

11. Faça um programa que leia um string (seqüência de caracteres) alfabético, utilizando a função **void LerString(char *s, int t, int tp)** e converta esta seqüência de caracteres toda para maiúsculas e minúsculas. Para realizar tal tarefa construa as seguintes funções, que são:

void StrMai(char *s), que recebe como parâmetro o endereço do primeiro byte de uma seqüência de caracteres (string) e converte-os para maiúsculas.

void StrMin(char *s), que recebe como parâmetro o endereço do primeiro byte de uma seqüência de caracteres (string) e converte-os para minúsculas.

char ChrMai(char c), que recebe um caractere, convete-o para maiusculo e retorna este caractere convertido.

char ChrMai(char c), que recebe um caractere, convete-o para minúsculo e retorna este caractere convertido.

12. Construa um programa que utilize a função **void LerString(char *s, int t, int tp)**, para ler uma seqüência de caracteres pelo teclado. O programa deve mostrar o tipo do string (da seqüência de caracteres), que pode ser: **0 - ALFABÉTICO** (só caracteres alfabéticos), **1 - ALFANUMÉRICO** (só caracteres alfabéticos e ou numéricos), **2 - NUM_INTEIRO** (só caracteres numéricos), **3 - NUM_REAL** (só caracteres numéricos e o caractere ponto) e **4 - SEM_TIPO** (qualquer caractere ASCII). Para realizar tal tarefa construa um função que recebe o endereço de um string e retorna um inteiro conforme o tipo do string. O protótipo da função seria: **int TipoString(char *s)**.

13. Dado o código abaixo:

```
#define TAM 20

void MemSet(char *, char, int);

void main()
{
        char *s = (char *) malloc(TAM);     // alocação de memória (heap)

        if (!s)
        {
                puts("Faltou memória . . .\a\a");
                exit(0);
        }
        puts(s);
        MemSet(s,'v',50);
        puts(s);
        free(s);   // liberar a memória reservada com malloc
        getch();
}

void MemSet(char *s, char c, int t)
{
        ________________________________________
        ________________________________________
        ________________________________________
        ________________________________________
        ________________________________________
        ________________________________________
        ________________________________________
}
```

Construa o código da função MemSet, que preenche uma área de memória de t bytes, com o caractere c.

14. Dado o programa abaixo

```
struct FONE
{
        char Código[3];
        char Área[4];
        char Número[4];
};

void MostrarFone( _______________, _______________ );

void main()
{
        char s[11] = "0413301515";
        struct FONE f = {{041},{330},{1563}};

        MostrarFone(s,11);
        MostrarFone(&f,11);
        getch();
}

void MostrarFone( _______________, _______________ );
{
        ______________________________________________
        ______________________________________________
        ______________________________________________
        ______________________________________________
        ______________________________________________
        ______________________________________________
        ______________________________________________
        ______________________________________________
        ______________________________________________
        ______________________________________________
        ______________________________________________
        ______________________________________________
}
```

Cosntrua a função MostrarFone. Esta função mostra o número de um telefone, que pode estar em um string ou em uma estrutura, como é percebido no código acima. O número do telefone é mostrado no seguinte formato: (041) 330-1515.

15.2 OPERADORES DE BITS

1. Complete a função main do código que segue implementando as funções que estão sendo chamadas. O bit mais significativo seria o bit de número 8 (oito) e o bit menos significativo seria o bit de número 1 (um), ou seja, a númeração dos bits vão da direita para a esquerda (menos significativo para o mais significativo).

Funções a serem implementadas:

ContarBitsLigado - que recebe o conteúdo de um caractere como parâmetro e retorna o número de bits ligados deste caractere.

DesligarBit - que recebe o endereço de um caractere e o número do bit a ser desligado.

InverterBit - que rebebe o endereço de um caractere e o número do bit a ter o seu valor invertido, se 1 (um) tornar 0 (zero) e vice-versa.

LigarBit - que recebe o endereço de um caractere e o número do bit a ser ligado.

EstadoBit - que recebe o conteúdo de um caractere e o bit a ser verificado o estado. Ligado ou desligado.

```
/******************************************************************************
        Protótipos de funções
******************************************************************************/

_______________ ContarBitsLigado( _______________, _______________);
_______________ DesligarBit( _________________, _________________);
_______________ InverterBit( _________________, _________________);
_______________ LigarBit( _________________, _________________);
_______________ EstadoBit( _________________, _________________);

/******************************************************************************
        Funções
******************************************************************************/

void main()
{
        unsigned char Chr, Nbits, Bit;

        Chr = 0x80;
        LigarBit(&Chr,4);
        Nbits = ContarBitsLigado(Chr);
        LigarBit(&Chr,5);
        Nbits = ContarBitsLigado(Chr);
        Chr = ~Chr;
        DesligarBit(&Chr,6);
        DesligarBit(&Chr,5);
        Nbits = ContarBitsLigados(Chr);
        InverterBit(&Chr,4);
        Bit = EstadoBit(Chr,1);
        DesligarBit(&Chr,1);
```

```
        DesligarBit(&Chr,6);
        Nbits = ContarBitsLigados(Chr);
}

_______________ ContarBitsLigado( _______________, _______________)
{
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
}

________________ DesligarBit( _________________, _________________)
{
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
}
```

________________ InverterBit(________________, ________________)
{
____
____
____
____
____
____
____
____
}

________________ LigarBit(________________, ________________)
{
____
____
____
____
____
____
____
____
}

________________ EstadoBit(________________, ________________)
{
____
____
____
____
____
____
____
____
}

2. Complete a função main do código que segue implementando as funções que estão sendo chamadas. O bit mais significativo seria o bit de número 8 (oito) e o bit menos significativo seria o bit de número 1 (um), ou seja, a númeração dos bits vão da direita para a esquerda (menos significativo para o mais significativo).

Funções a serem implementadas:

ContarBitsLigado - que recebe o conteúdo de um caractere como parâmetro e retorna o número de bits ligados deste caractere.

DesligarBit - que recebe o conteúdo de um caractere e o número do bit a ser desligado e retorna o caractere com o bit desligado.

InverterBit - que rebebe um caractere e o número do bit a ter o seu valor invertido, se 1 (um) tornar 0 (zero) e vice-versa e retorna o caractere com o bit invertido.

LigarBit - que recebe o conteúdo de um caractere e o número do bit a ser ligado e retorna o caractere com o bit desligado.

EstadoBit - que recebe o conteúdo de um caractere e o bit a ser verificado o estado e retorna um (ligado) ou zero (desligado).

```
/*****************************************************************************
        Protótipos de funções
*****************************************************************************/

_______________ ContarBitsLigado( _______________, _______________);
_______________ DesligarBit( _________________, _________________);
_______________ InverterBit( _________________, _________________);
_______________ LigarBit( _________________, _________________);
_______________ EstadoBit( _________________, _________________);

/*****************************************************************************
        Funções
*****************************************************************************/

void main()
{
        unsigned char Chr, Nbits, Bit;

        Chr = 0x90;
        Chr = LigarBit(Chr,4);
        Nbits = ContarBitsLigado(Chr);
        Chr = LigarBit(Chr,5);
        Nbits = ContarBitsLigado(Chr);
        Chr = ~Chr;
        Chr = DesligarBit(Chr,6);
        Chr = DesligarBit(Chr,5);
        Nbits = ContarBitsLigados(Chr);
        Chr = InverterBit(Chr,4);
        Bit = EstadoBit(Chr,1);
```

```
    Chr = DesligarBit(Chr,1);
    Chr = DesligarBit(Chr,6);
    Nbits = ContarBitsLigados(Chr);
}

_______________ ContarBitsLigado( _______________, _______________)
{
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
}

________________ DesligarBit( _________________, _________________)
{
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
}
```

```
________________ InverterBit( ________________, ________________)
{
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
}

________________ LigarBit( ________________, ________________)
{
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
}

________________ EstadoBit( ________________, ________________)
{
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
        ____________________________________________________________
}
```

16. ARQUIVOS

O trabalho com arquivos é fundamental em programação. Através dos arquivos é que conseguimos fazer programas que conseguem produzir informações permanentes no computador. Informação estas que poderão ser alteradas e ou excluídas no futuro. Armazenando os dados em discos na forma de arquivos.

O conceito de arquivo está no nosso cotidiano, que é bastante normal a utilização de arquivos para armazenar-mos grandes quantidades de informação, por um grande período de tempo. Exemplos: os arquivos mantidos por uma companhia telefônica, agenda de telefones, as informações de um paciente em um consultório médico e etc.

O arquivo é um conjunto (coleção) de registros (de uma estrutura de dados). Cada registro ocupa uma posição fixa dentro do arquivo. Os registros são estruturas formadas por um conjunto de informações chamadas de campos.

Exemplo de um arquivo do cotidiano seria o arquivo de uma biblioteca, que possui o catálogo dos livros existentes na biblioteca.

Para isto temos a ficha do livro:

Código do livro: ___________________________
Título: ___________________________________
Autor: ___________________________________
Assunto: _________________________________
Editora: ________________________ Ano: _____

Cada ficha de um livro seria um registro e cada registro possui seis campos que são: código do livro, título, autor, assunto, editora e ano.

Quando todas as fichas (registros) dos livros forem colocadas juntas em um mesmo local (um arquivo de aço) formando assim o arquivo da biblioteca, podendo ser manipulado pelas pessoas, que utilizam o arquivo de fichas.

Código do livro: 13.546
Código do livro: 13.002
. . . .
. . . .
Código do livro: 12.932
Código do livro: 12.903
Título: Programando em linguagem C
Autor: José João Krigüer
Assunto: Programação de computadores
Editora: Céu Azul Ltda Ano: 1992

16.1 REPRESENTAÇÃO DE REGISTROS

Para representarmos um registro na codificação da linguagem C, lançaríamos mão de estruturas e uniões, quando necessário.

Em cima do exemplo anterior, a do arquivo da biblioteca, cada ficha de um livro constante em um arquivo de aço, onde todas as fichas são guardadas, cada ficha é o registro de um livro, que contém campos (código, título, autor, assunto, editora e ano). Para representarmos este registro na linguagem teríamos:

```
struct LIVRO
{
        int Código;
        char Título[35];
        char Autor[40];
        char Assunto[35];
        char Editora[30];
        int Ano;
};
```

A estrutura LIVRO é análoga ao registro Livro (ficha do arquivo), onde os seus elementos (código, título, autor, assunto, editora e ano) correspondem aos campos dos registros.

16.2 FLUXOS E ARQUIVOS

A linguagem C não possui comandos de entrada e saída. O trabalho todo relacionado à manipulação de arquivos se dá através de funções da biblioteca padrão. Este tipo de abordagem, garante ser um sistema bastante poderoso e flexível.

Na linguagem C existe dois métodos para o tratamento de arquivos. O primeiro o sistema ANSI, com buffer, também chamado de formatado ou alto nível e um segundo sistema UNIX, sem buffer, também chamado de não formatado.

O sistema de entrada e saída na linguagem C, torna os diversos detalhes dos dispositivos reais transparentes para o programador. Isto para dar maior portabilidade possível.

As funções de entrada e saída trabalham sobre fluxos de dados. Os fluxos podem ser conectados à dispositivos reais, sendo considerados como arquivos. Um fluxo é uma seqüência de dados e existem dois tipo:

Os fluxos de TEXTO, que são seqüências de caracteres ASCII, muitas vezes linhas terminadas com uma nova linha (\n).

Os fluxos BINÁRIOS, que são seqüências de bytes em estado bruto, não ocorrendo nenhuma tradução.

A maior parte das funções de entrada e saída são projetadas para operar sobre fluxos. Estes fluxos garantem a flexibilidade, podendo ser atribuídos à diferentes dispositivos reais. Algumas dessas conexões já estão construídas quando iniciado o programa. Cinco fluxos padronizados estão definidos e conectados aos vários dispositivos físicos que são:

Fluxo	Conectado
stdout	tela
stdin	teclado
stdprn	impressora
stdaux	tela
stderr	tela

Para manipularmos arquivos precisamos das seguintes etapas:

- abrir o arquivo.
- ler e ou gravar no arquivo.
- fechar o arquivo.

Abrir o arquivo, significa criarmos um fluxo (texto ou binário) para podermos em seguida gravar e ou ler dados no arquivo, através do dispositivo de fluxo criado. Após a manipulação deve ser fechado o arquivo, fechar o fluxo criado anteriormente. Para realizarmos tais tarefas com arquivos, existe na biblioteca padrão funções que serão brevemente comentadas na seqüência.

Para abrirmos um arquivo é necessário criarmos um fluxo e este fluxo é criado através da declaração de um ponteiro para uma estrutura já definida na biblioteca padrão (STDIO.H). Que segue a título de curiosidade a sua definição:

File control structure for streams.

```
typedef struct
{
        short           level;
        unsigned        flags;
        char            fd;
        unsigned char   hold;
        short           bsize;
        unsigned char   *buffer, *curp;
        unsigned        istemp;
        short           token;
} FILE;
```

_F_xxxx

File status flags of streams

Name	Meaning
_F_RDWR	Read and write
_F_READ	Read-only file
_F_WRIT	Write-only file
_F_BUF	Malloc'ed buffer data
_F_LBUF	Line-buffered file
_F_ERR	Error indicator
_F_EOF	EOF indicator
_F_BIN	Binary file indicator
_F_IN	Data is incoming
_F_OUT	Data is outgoing
_F_TERM	File is a terminal

Então antes de abrirmos um fluxo de arquivo temos a declaração:

```
FILE *f;
```

Onde f é um ponteiro para um tipo definido de estrutura FILE. Vale lembrar que nós não nos preocupamos em saber muito sobre FILE e seus sinalizadores de estados (status flags), estas informações são transparentes para o programador, sendo que a completa e total manipulação deste ficam a cargo, internamente, das funções da biblioteca padrão, que trata de arquivos.

Após a declaração do ponteiro do tipo FILE, devemos abrir o arquivo utilizando a função fopen, que possui o seguinte protótipo:

16.3 FOPEN(...)

FILE *fopen(**char** *, **char** *);

Onde os parâmetros passados são o nome do arquivo (com ou sem caminho) e o modo, que determina como o arquivo vai ser aberto (leitura, escrita, leitura e ou escrita).

Modo	Significado
r	Abre um arquivo texto para leitura
w	Cria um arquivo texto para escrita
a	Anexa a uma arquivo texto
rb	Abre um arquivo binário para leitura
wb	Cria um arquivo binário para escrita
ab	Anexa a um arquivo binário
r+	Abre um arquivo texto para leitura
w+	Cria um arquivo binário para escrita
a+	Anexa ou cria um arquivo texto para leitura e ou escrita
r+b	Abre um arquivo binário para leitura e ou escrita
w+b	Cria um arquivo binário para leitura e ou escrita
a+b	Anexa a um arquivo binário para leitura e ou escrita

```
f = fopen(“TESTE.$$$”,”w”);
```

A função fopen retorna o endereço de uma estrutura FILE (retorna um ponteiro). Este valor nunca deve ser alterado. Caso aconteça algum erro na abertura do arquivo (não existe o diretório, nome do arquivo errado e etc.) a função fopen retornará NULL (nulo, zero). Devido ao retorno de fopen, deve sempre ser feito um teste para a verificação da abertura do aquivo.

```
if (!f)
{
        puts(“Problemas na abertura do arquivo . . . \a”);
        exit(0);
}
```

Já que não conseguiu abir o arquivo, dá-se uma mensagem e sai do programa, neste caso.

O código escrito anteriormente, poderia ser rescrito de uma outra maneira, utilizando a função fopen já no comando de seleção, assim:

```
if (!(f = fopen(“TESTE.$$$”,”a”)))
{
        puts(“Problemas na abertura do arquivo . . . \a”);
        exit(0);
}
```

Ler e ou gravar no arquivo, isto significa apanhar ou por uma informação (um número de bytes) em um arquivo que esteja aberto, informações.

16.4 FREAD(...)

Para ler informações de um arquivo, podemos utilizar a função fread, que possui protótipo:

size_t fread(**void** *ptr, size_t size, size_t n, FILE *stream);

Esta função lê um número especifico de ítens de dados de igual tamanho de um fluxo aberto (arquivo aberto).

Os argumentos da função são:

ptr – Aponta para o bloco no qual os dados serão lidos.

size – Tamanho de cada ítem que será lido, em bytes.
n – Número de ítens que serão lidos.
stream – Aponta para o fluxo aberto (arquivo).

O número de bytes lidos no total é igual à (n * size).

Retorna o número de ítens lidos (se conseguiu ler) ou um final de arquivo ou um erro, retornando possivelmente zero.

Exemplo:

```
1.  void main(void)
2.  {
3.      FILE *stream;
4.      char msg[] = "Isto é um teste";
5.      char buf[20];
6.
7.      if (!(stream = fopen("DUMMY.FIL", "w+")))
8.      {
9.              puts("Não foi possível abrir o aquivo . . .\a\a\n");
10.             return;
11.     }
12.     /* escreve algum dado no arquivo */
13.     fwrite(msg,strlen(msg) + 1,1,stream);
14.     /* posiciona no inicio do arquivo */
15.     fseek(stream, SEEK_SET, 0);
16.     /* lê o dado e mostra no monitor de vídeo */
17.     fread(buf,strlen(msg) + 1,1,stream);
18.     printf("%s\n", buf);
19.     /* fecha o arquivo */
20.     fclose(stream);
21. }
```

16.5 FWRITE(...)

Para gravar informações em um arquivo, podemos utilizar a função fwrite, que possui protótipo:

size_t fwrite(const **void** *ptr, size_t size, size_t n, FILE*stream);

Esta função escreve um número especifico de ítens de dados de igual tamanho de um fluxo aberto (arquivo aberto).

Os argumentos da função são:

ptr – Aponta para o bloco no qual os dados serão escritos.
size – Tamanho de cada ítem que será escrito, em bytes.
n – Número de ítens que serão escritos.
stream – Aponta para o fluxo aberto (arquivo).

O número de bytes escritos no total é igual à (n * size).

Retorna o número de ítens escritos (se conseguiu escrever) ou um erro possivelmente zero.

Exemplo:

```
1.  struct INFO
2.  {
3.      int i;
4.      char ch;
5.  };
6.  void main(void)
7.  {
8.      FILE *stream;
9.      struct INFO s;
10.
11.     /* abre o arquivo TEST.$$$ */
12.     if (!(stream = fopen("TEST.$$$", "wb")))
13.     {
14.             puts("Não foi possível abrir o aquivo . . .\a\a\n");
15.             return;
16.     }
17.     s.i = 0;
18.     s.ch = 'A';
19.     fwrite(&s,sizeof(s),1,stream); /* escreve a estrutura s no arquivo */
20.     fclose(stream); /* fecha o arquivo */
21. }
```

16.6 FCLOSE(...)

Esta função fecha um fluxo de arquivo. Seu protótipo é:

int fclose(FILE *stream);

Todos os buffers associados com a fluxo são esvaziados antes de serem fechados.

Se houver sucesso no fechamento do fluxo de arquivo, retorna 0. Se ocorrer erro retorna um EOF.

Exemplo:

```
1.  void main(void)
2.  {
3.      FILE *fp;
4.      char buf[11] = "0123456789";
5.
6.      /* cria um arquivo contendo dez bytes */
7.      fp = fopen("DUMMY.FIL", "w");
8.      if (!fp)
9.      {
10.             puts("Não foi possível abrir o aquivo . . .\a\a\n");
11.             return;
12.     }
13.     fwrite(&buf, strlen(buf), 1, fp);
14.     /* fecha o arquivo */
15.     fclose(fp);
16. }
```

16.7 FTELL(...)

Retorna a posição corrente do ponteiro de um arquivo de fluxo. O protótipo seria:

long ftell(FILE *stream);

Se o arquivo é binário, o desvio é medido em bytes do inicio do arquivo.

O valor retornado por ftell pode ser usado na chamda subseqüênte de um fseek. Retorna a posição corrente do ponteiro de um fluxo de arquivo. Se ocorrer erro, retorna –1L.

Exemplo

```
1.  void main(void)
2.  {
3.      FILE *stream;
4.
5.      stream = fopen("MYFILE.TXT", "w+");
6.      if (!stream)
7.      {
8.          puts("Não foi possível abrir o aquivo . . .\a\a\n");
9.          return;
10.     }
11.     fprintf(stream, "This is a test");
12.     printf("The file pointer is at byte %ld\n", ftell(stream));
13.     fclose(stream);
14. }
```

16.8 FSEEK(...)

Reposiciona o ponteiro associado à um fluxo de arquivo para a nova posição. O protótipo seria:

int fseek(FILE *stream, long offset, **int** whence);

Os arqgumentos são:

- stream - Ponteiro para um fluxo de arquivo.
- offset – Diferença em bytes entre de onde e a nova posição. Para fluxos de arquivos no modo texto, offset deverá ser 0 ou um valor retornado por ftell.
- whence – Uma das três SEEK_SET (inicio), SEEK_CUR (corrente) e SEEK_END (final) posições de localização do ponteiro de fluxo do arquivo. (0, 1, ou 2 respectivamente).

Depois de um fseek, a próxima operação de atualização de um arquivo pode ser uma leitura ou escrita.

O retorno seria: O ponteiro é movido com sucesso, fseek retorna 0. Caso não consiga mover o ponteiro retorna um valor não zero (ocorreu algum erro)..

Exemplo:

```
1.  long FileSize(FILE *stream);
2.
3.  void main(void)
4.  {
5.      FILE *stream;
6.      stream = fopen("FILE.TXT", "w+");
7.      if (!stream)
8.      {
9.          puts("Não foi possível abrir o aquivo . . .\a\a\n");
10.         return;
11.     }
12.     fprintf(stream, "Isto é um teste ");
13.     printf("O tamanho do arquivo FILE.TXT é %ld bytes\n", FileSize(stream));
14.     fclose(stream);
15. }
```

```
16.  long FileSize(FILE *stream)
17.  {
18.        long curpos, length;
19.
20.        curpos = ftell(stream);
21.        fseek(stream, 0L, SEEK_END);
22.        length = ftell(stream);
23.        fseek(stream, curpos, SEEK_SET);
24.        return (length);
25.  }
```

16.9 FEOF(...)

É uma macro que testa se o final de um arquivo de fluxo foi alcançado. O protótipo é:

int feof(FILE *stream);

Retorna um não zero se um final de arquivo foi encontrado. Retorna um zero se o final de arquivo não foi encontrado.

Exemplo:

```
1.   void main(void)
2.   {
3.         FILE *stream;
4.
5.         /* abre um arquivo para leitura */
6.         stream = fopen("DUMMY.FIL", "r");
7.         if (!stream)
8.         {
9.               puts("Não foi possível abrir o aquivo . . .\a\a\n");
10.              return;
11.        }
12.        /* lê um charactere de um arquivo */
13.        fgetc(stream);
14.        /* verifica se é EOF */
15.        if (feof(stream))
16.              puts("É final de arquivo ");
17.        fclose(stream);
18.  }
```

16.10 FGETS(...) e FPUTS(...)

A função fgets lê uma seqüência de caracteres (string) de um fluxo de arquivo. A função fputs escreve uma seqüência de caracteres (string) em um fluxo de arquivo.

O seus protótipos são:

char *fgets(**char** *s, **int** n, FILE *stream);
int fputs(const **char** *s, FILE *stream);

A função fgets lê caracters de um fluxo de arquivo, parando quando **for** lido também um n – 1 caracteres ou um caractere de linha nova ('\n'), o que vier primeiro. A função fgets coloca um byte nulo ('\0') depois do caractere nova linha, indicando o final da seqüência de caracteres (string).

A função fputs copia uma seqüência de caracteres (string), terminado com o caractere nulo para um fluxo de arquivo, não colocando um caractere de nova linha e nem copiando o caractere nulo.

A função fgets retorna o um ponteiro para s. Encontrando um EOF ou ocorrendo um erro fgets retorna NULL.

A função fputs retorna retorna o último caractere escrito. Se ocorrer algum erro fputs retorna EOF.

Exemplo:

```
1.  void main(void)
2.  {
3.      FILE *stream;
4.      char string[16] = "Isto é um teste ";
5.      char msg[20];
6.
7.      /* abre um arquivo para leitura */
8.      stream = fopen("DUMMY.FIL", "w+");
9.      if (!stream)
10.     {
11.             puts("Não foi possível abrir o aquivo . . .\a\a\n");
12.             return;
13.     }
14.     /* escreve um string em um arquivo */
15.     fwrite(string,strlen(string),1,stream);
16.     /* posiciona no início do arquivo */
17.     fseek(stream, 0, SEEK_SET);
18.     /* lê um string de um arquivo */
19.     fgets(msg,strlen(string) + 1,stream);
20.     /* mostra o string lido */
21.     printf("%s", msg);
22.     fclose(stream);
23. }
```

Exemplo:

```
1.  void main(void)
2.  {
3.      FILE *stream;
4.      int i = 100;
5.      char c = 'C';
6.      float f = 1.234;
7.
8.      /* abre um arquivo para leitura */
9.      stream = fopen("DUMMY.FIL", "w+");
10.     if (!stream)
11.     {
12.             puts("Não foi possível abrir o aquivo . . .\a\a\n");
13.             return;
14.     }
15.     /* escreve algum dado no arquivo */
16.     fprintf(stream, "%d %c %f", i, c, f);
17.     /* fecha um arquivo */
18.     fclose(stream);
19. }
```

16.11 EXERCÍCIOS PROPOSTOS

1. Faça um programa, em linguagem C, que leia caracteres a caractere do teclado, utilizando a função **getchar** e grave em um arquivo com o nome **KTOD.DAT**, através da função **fputc**. O programa termina quando o caractere lido for um **$**.

2. Faça um programa, em linguagem C, que leia caractere a caractere do arquivo **KTOD.DAT**, através da função **fgetc** e mostrando o caractere através da função **putchar**.

3. Faça um programa, em linguagem C, que gere o arquivo **LADOS.DAT**. Este arquivo possui 3000 valores inteiros, gerados aleatoriamente utilizando a função **random**.

4. Faça um programa, em linguagem C, que leia o arquivo **LADOS.DAT**, utilizando a estrutura definida abaixo:

```
struct TRIANGULO
{
        int a, b, c;
};
```

O programa deverá informar quantos são triângulos e não triângulos.

5. Faça um programa, em linguagem C, que leia o arquivo **LADOS.DAT** e utilizando uma estrutura TRIANGULO, informe qual o triângulo, que possui o maior perímetro.

6. Faça um programa, em linguagem C, que gere o arquivo **TRIAN.DAT**, que possui todos os lados que formam um triângulo, extraídos do arquivo **LADOS.DAT**.

7. Considere o fragmento de código abaixo:

```
/***************************************************************************
        Includes
***************************************************************************/

#include <conio.h>
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <lib.h>

/***************************************************************************
        Estruturas
***************************************************************************/

struct DATA
{
        char    Dia;
        char    Mes;
        int     Ano;
};

struct NOME
{
        char    Nome[40];
        char    Sexo;
        struct DATA Data; // data de nascimento
};
```

```
struct FONE
{
	char Codigo[3];
	char Prefixo[4];
	char Numero[4];
};

struct ENDERECO
{
	char Rua[35];
	char Bairro[25];
	char Cep[8];
};

struct REG
{
	char Codigo[6]; // seqüência de caracteres numéricos
	struct NOME Nome;
	struct FONE Fone;
	struct ENDERECO Endereco;
	struct DATA Data; // data do sistema operacional
	char Email[40];
};

/*****************************************************************************
	Prototipos de funções
*****************************************************************************/

void ApanharDataSistema(struct DATA *Data);
void LerRegistro(struct REG *);
void GerarCodigo(char *);
void GerarEmail(struct REG *, char *);
void GravarRegistro(struct REG);

/*****************************************************************************
	Funcoes
/*****************************************************************************/

void main()
{
	struct REG Reg;

	randomize(); // função para inicializar o gerador de números aleatórios
	do
	{
		LerRegistro(&Reg);
		if (!Reg.Nome.Nome[0])
			break;
		GravarRegistro(Reg);
	} while (1);
}

/*****************************************************************************
	Sintaxe: void LerRegistro(struct REG *Reg)
	Função: Ler uma estrutura do tipo REG.
	Entrada: Ponteiro para uma estrutura do tipo REG.
	Saída:   Nenhuma.
/*****************************************************************************/

void LerRegistro(struct REG *Reg)
{
```

```
        GerarCodigo(Reg->Codigo,sizeof(Reg->Codigo));
        . . .     . . .     . . .     . . .
        // codigo para a leitura dos outros elementos da estrutura REG
        . . .     . . .     . . .     . . .
        ApanharDataSistema(Reg->Data);
        GerarEmail(Reg,"system");
}

/***************************************************************************
        Sintaxe: void ApanharDataSistema(struct DATA *Data)
        Função: Apanhar a data do sistema operacional.
        Entrada: Ponteiro para uma estrutura do tipo DATA.
        Saída:   Nenhuma.
***************************************************************************/

void ApanharDataSistema(struct DATA *Data)
{
        . . .     . . .     . . .     . . .
}

/***************************************************************************
        Sintaxe: void GravarRegistro(struct REG *Reg)
        Função: Gravar em disco o conteudo de uma struct Reg, no
                  arquivo CADASTRO.DAT.
        Entrada: Ponteiro para uma estrutura do tipo REG.
        Saída:   Nenhuma.
***************************************************************************/

void GravarRegistro(struct REG Reg)
{
        . . .     . . .     . . .     . . .
}

/***************************************************************************
        Sintaxe: void GerarEmail(struct REG *Reg, char *Provedor)
        Função: Gerar o email a partir do nome completo da pessoa.
        Entrada: Ponteiro para uma estrutura do tipo REG e um char.
        Saída:   Nenhuma.
***************************************************************************/

void GerarEmail(struct REG *Reg, char *Provedor)
{
        . . .     . . .     . . .     . . .

}

/***************************************************************************
        Sintaxe: void GerarCodigo(char *Codigo, int Tamanho)
        Função: Gerar um código numérico (caracteres numéricos).
        Entrada: Ponteiro para um char (Código) e o tamanho.
        Saída:   Nenhuma.
***************************************************************************/

void GerarCodigo(char *Codigo, int Tamanho)
{
        . . .     . . .     . . .     . . .
}
```

Complete o programa com o código necessário para as funções que faltam ser implementadas.

LerRegistro, esta função deve ler um tipo **struct** REG, passado por referência para a função.

GerarCodigo, está função gera um string preenchido aleatóriamente com caracteres numéricos. Utilize para isto a função random.

ApanharDataSistema, para a implementação desta função utilize a função **getdate**.

GerarEmail, o email deve ser gerado a partir do nome completo da pessoa, pegando sempre a letra inicial de cada nome do nome, mais um caractere gerado aleatóriamente. Ex.: José Nogueira Filho, o emil poderia ser jnfk@system.com.br.

GravarRegistro, função que grava em disco o conteúdo de uma variável do tipo **struct** REG, no arquivo **CADASTRO.DAT**.

8. Faça um programa, em linguagem C, que extraia do arquivo **CADASTRO.DAT** os campos código e email, gerando um outro arquivo, com estes campos, o novo arquivo deve ter o nome de **EMAIL.DAT**.

9. Faça um programa, em linguagem C, que retire do arquivo **CADASTRO.DAT** o campo email, reorganizando o arquivo todo. Faça tudo isto no mesmo arquivo.

17. GLOSSÁRIO

ANSI - American National Standards Institute (Instituto Nacional de Padronização Americano).

ARQUIVO (FILE) - Uma coleção organizada de informações, preparada para ser usada com alguma finalidade. Os registros em um arquivo podem ser seqüenciais ou não, dependendo das necessidades de sua utilização.
Uma coleção de registros tratada como uma unidade.
Um conjunto de registros inter-relacionados tratados como uma unidade por exemplo, em controle de estoque o arquivo deve ser constituído de itens (registros) de estoque.
Unidade principal de dados físicos formada por um conjunto de registros físicos dispostos em formatos prescritos e descrita pela informação de controle a qual o sistema tem acesso.

ASCII - American National Standard Code **for** Information Interchange (Código para Intercâmbio de Informação Nacional Padrão Americano), X 3.4.1968. Um código de nível 8 (7 bits e paridade) desenvolvido por um subcomitê da USASI. Este código foi desenvolvido com a finalidade de padronizar as comunicações americanas.

ASSEMBLE - Montar, preparar um programa em linguagem objeto a partir de um programa em linguagem simbólica, substituindo os códigos de operações simbólico e absoluto ou endereço relocável para endereço simbólico.

ASSEMBLER - Montador, programa que monta; transforma os códigos de operação simbólicos em códigos absolutos ou de máquina.

ASSEMBLY - Montagem, saída produzida pelo montador ASSEMBLER.

BIT - Abreviatura de "binary digit". É a menor unidade de um computador. Magnetizado, no estado desligado assume o valor zero e no estado ligado assume o valor um.

BCPL - Linguagem de programação de sistemas que incorpora as estruturas de controle necessárias à programação estruturada. Sua principal característica consiste em ser uma linguagem de tipo livre; ou seja, o único tipo de objeto de dados que pode ser usado é uma palavra composta de bits.

CRIPTOGRAFIA - Proteção de uma mensagem mediante em dos métodos (código cifrado) que transformam um texto em linguagem natural em texto cifrado ou vice-versa. O primeiro ou método primário, consiste em substituição que cada elemento individual do texto cifrado em código, por ser elemento correspondente em linguagem natural. A lista dessas substituições recebe o nove de código de chaves e deve manter-se secreto com o objetivo de proteger a informação. O processo de cifragem consiste em mudar um texto em linguagem natural para texto em linguagem cifrada (criptografada) mediante transformação criptográfica, geralmente de tal modo que cada bit, caractere ou palavra do texto normal (seja substituído por bit, caractere ou palavra do texto cifrado (criptografado).
A transformação do dado ou informação, para encobrir ou dissimular seu significado.

DRIVE - Mecanismo impulsionador.

PALAVRA - (WORD) Conjunto ordenado de bytes (caracteres) que ocupa um localização no armazenamento e é tratado pelos circuitos do computador (em operações de transferências, aritmética etc.), com uma unidade.
Ordinariamente a palavra é tratada, tanto pela unidade de controle como pela unidade aritmética como uma quantidade.
O comprimento da palavra poder ser fixo ou variável dependendo do computador.

REGISTRADORES Dispositivo do hardware usado para armazenar uma certa quantidade de bit, bytes ou caracteres. Um registrador é normalmente construído de elementos tais como transmissores ou tubos e geralmente armazena uma palavra de informação. A programação comum exige que o registrador tenha condições de operar a informação e não apenas somente armazená-la.

REGISTRO (RECORD) - Um grupo de fatos relacionados ou campos de informações tratadas como uma unidade ou ainda uma lista de informações em uma forma a ser impressa.

ULA - Unidade lógico aritmética (ALU - Arithmetic Logical Unit). É a parte do hardware do computador na qual são realizadas as operações aritméticas e lógicas.
A unidade aritmética geralmente é constituída de um acumulador, alguns registradores especiais para o armazenamento dos operandos e resultados, suplementados por circuitos de deslocamentos e de seqüência, divisão e outras operações desejadas.

18. BIBLIOGRAFIA

C **for** engineers, ***Brian Bramer & Susan Bramer***. Ed. John Wiley & Sons.

TURBO C - GUIA DO USUÁRIO. ***Herbert Schildt***. Ed. Makron Books.

TREINAMENTO EM LINGUAGEM C, CURSO COMPLETO - Módulos 1 e 2. ***Victorine, Viviane e Mizrahi.*** Ed. McGraw-Hill.

Dominando o Turbo C. ***Stan Kelly-Bootle***. Ed. Ciência Moderna.

Linguagem C - ***Kernighan & Ritchie***

C, Completo e total - ***Herbert Schildt.*** Ed. Makron Books.