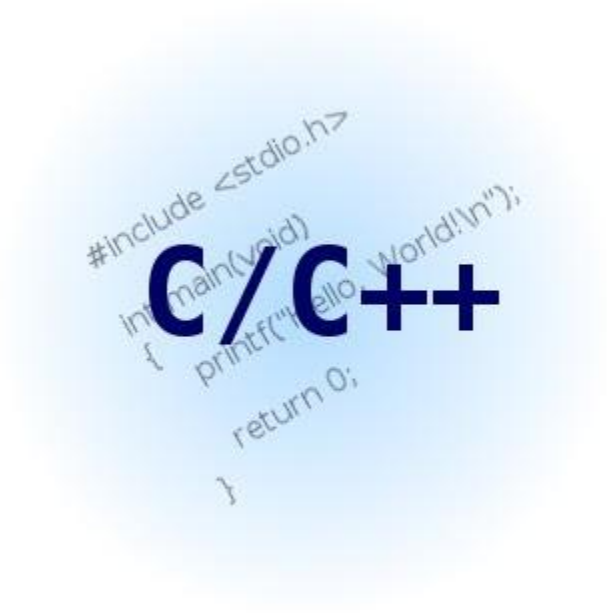
Linguagem C


#include <stdio.h>
int main(void)
{
printf("Hello, World!\n");
return 0;
}
C/C++

Responsáveis

A apostila de **Linguagem C** é de responsabilidade do **Programa de Educação Tutorial** do curso de **Engenharia Elétrica** da **Universidade Federal do Ceará**, tendo como principais responsáveis os bolsistas:

- Alexcya Lopes Alexandre
- Camila Tavares Vitoriano
- Felipe Carvalho Sampaio
- Maria Yasmin Almeida Sampaio
- Nestor Rocha Monte Fontenele
- Roberto Aaron Marques Braga
- Túlio Naamã Guimarães Oliveira

SUMÁRIO

Introdução a C

O início de C

A programação como um todo teve como inicio a necessidade de otimização de processos, para conseguirmos reduzir o tempo de determinada tarefa e principalmente automatizá-la, pois a máquina não possui o fator humano que influencia no processo de execução de qualquer tarefa.

Um dos primeiros programadores que se tem noticia foi uma mulher, Ada Lovelace, filha de Lord Byron, em 1842 programou algoritmos que serviriam para resolver problemas matemáticos na máquina de computar inventada por Charles Babbage.

Em 1801, Joseph Marie Jacquard projetou uma maquina de tecelagem que tivesse varias opões de cor para os tecidos, ele fez isso através de cartões em que cada tipo de perfuração representava uma cor, assim podemos dizer que ela possuía um algoritmo. Através dessa mesma principio, só que aplicado a passagem de trens, onde o condutor observava onde era o assento de cada passageiro através das perfurações dos cartões. Com isso, Heman Hollerith projetou uma máquina que lia as perfurações dos cartões, mas para otimizar o senso americano, que precisou de bem menos tempo para averiguação dos resultados.

Em 1940 foi feito um dos primeiros computadores elétricos, muito arcaico e caro e difícil de programar, logo surgiram algumas linguagens de programação como o C-10 e o ENIAC. Anos mais tardes, foram inventados o COBOL, o LISP e o FORTAN que avançaram na busca da programação que temos hoje.

Por volta dos anos 70 surge a linguagem C, através de mudanças feitas em uma linguagem anteriormente denominada de B. Ela era fácil de programar, se comparada a outras linguagens como Assembly. Foi criada em um dos laboratórios de Gran Bell, o inventor do telefone, por Ken Thompson e Dennis Ritchie.

Nivéis de linguagem

Temos vários tipos de linguagem de programação e o que define o nível de uma linguagem é a sua proximidade com a linguagem interpretada pelo processador, ou seja quanto mais próxima deste menor o nível.

As linguagens de baixo nível são as chamadas linguagens de máquina, que possuem apenas sequências de bits em strings e tornava inviável a interpretação e programação do computador. Por isso foi inventado o Assembly que possuía alguns comandos que ajuda na interpretação do código pelo programador e facilitou a programação em linguagem de máquina.

Para trazer a linguagem de programação cada vez mais para nosso cotidiano, foram criadas as linguagens de programação de alto nível que utilizam expressões do cotidiano como se e enquanto que torna mais fácil a elaboração do código. São exemplos desta a linguagem C, C++, Pascal e JAVA.

As vantagens da linguagem de baixo nível em relação as de alto nível é que um código, em Assembly por exemplo, ocupa menos espaço é mais rápido e também ajuda ter total controle do Hardware, porém a desvantagem é o número reduzido de comandos que dificulta sua utilização.

- **C x C++**

As duas linguagens são muito parecidas, sendo que o C++ é uma evolução do C. O C é uma linguagem estruturada, ou seja que tem uma estrutura padronizada de lógica, é usado em larga escala para programação de sistemas operacionais como o Linux e é rápido e ocupa menos espaço.

Porém quando estamos trabalhando com códigos muito extensos, o C torna se muito inviável, dai a necessidade de usarmos o C++, pois como é uma linguagem orientada a objetos, ao invés de repetirmos várias vezes uma mesma parte de um algoritmo podemos simplesmente criar uma classe para aquela parte, reduzindo as linhas do programa, assim tornando-o mais rápido.

Linguagem C

Inicialização de um programa

Todos os programas em C apresentam a seguinte estrutura básica:

```
#include <nome da biblioteca>

<tipo dos dados de entrada> main (<tipo dos dados de saída>)

{

        bloco de comandos;

}
```

Bibliotecas

Bibliotecas são arquivos que contêm inúmeras funções, as quais são inclusas no programa com a utilização da diretriz #include, como você pode observar na primeira linha do programa acima.

Observe abaixo uma lista das bibliotecas mais utilizadas:

- **<stdio.h>** : Ela contém funções que controlam entrada e saída de dados (o nome da biblioteca já sugere sua função: stdio : io = input e output = entrada e saída). As principais funções nela contidas são printf()e scanf(), as quais explicaremos melhor mais adiante;

- **<stdlib.h>** : Ela contém funções de alocação de memória, conversão de números para textos e vice-versa, entre outras funções;

- **<math.h>** : Essa biblioteca contém funções matemáticas em geral, tais como trigométricas e exponenciais. Vale ressaltar que não é necessário utilizá-la para executar funções que envolvem apenas as quatros operações fundamentais da matemática. São exemplos de funções nela contidas: sin(x)e sqrt();

- **<string.h>** : Ela contém funções de manipulação de cadeias de caracteres (strings).

- **<conio.h>** : Ela contém as funções getch e getch que têm a função de ler e exibir na tela respectivamente.

Lembre-se de que o C é *case sensitive*, ou seja, letras maiúsculas e minúsculas são lidas como caracteres diferentes. Os nomes de todas as bibliotecas devem ser escritos com letras minúsculas para que o compilador seja capaz de identificá-las.

Entrada e saída de dados

Os comandos de entrada são utilizados para receber os dados fornecidos pelo usuário, alocando-os em variáveis. Os mais utilizados são *cin*, *gets* e *scanf*.

O comando *cin* não deve ser usado para armazenar variáveis com caracteres em branco, pois ele só armazena até o último caractere que antecede o espaço em branco, sendo os caracteres posteriores desprezados.

Os comandos *scanf* e *gets* armazenam caracteres até ser pressionado o botão *Enter*. Sendo o comando *gets* o mais indicado para armazenar variáveis com caracteres em branco.

Exemplos:

- cin >> x; /* a variável 'x' receberá o valor digitado pelo usuário */
- scanf("%d",&x); /* a variável inteira 'x' receberá o valor digitado pelo usuário*/
- gets(ENDERECO); /* os caracteres digitados serão armazenados na variavel *gets* */

Os comandos de saída são utilizados para exibir os dados na tela ou na impressora, sendo os mais utilizados os comandos *printf* e *cout*.

Exemplos:

- printf("%d", x); /*exibe o valor contido na variavel inteira 'x'*/
- printf("%a.bf", X); /*exibe o numero real armazenado na variável com a casas para a parte inteira e b casas para a parte decimal. Por exemplo: "2.3%f" são duas casa para representar o inteiro e três casas para o decimal*/
- printf("O resultado eh: %2.3f",x); /* exibe a frase "O resultado eh" e, em seguida, o valor armazenado na variavel x */
- cout << A; /*exibe o valor contido na variável A*/
- cout << "O resultado eh" << A; /*exibe a frase entre parênteses e, em seguida, o valor contido na variável A*/

Tipos de dados e tamanhos

A Linguagem C apresenta cinco tipos básicos de dados, são eles:

- *char* : para caracteres (tamanho de 6 bytes);
- *int* : para números inteiros (tamanho de 2 bytes);
- *float* : para números reais (inteiros com muitos dígitos e números com casas decimais com tamanho de 4 bytes);
- *double* : utilizado quando se deseja obter uma precisão dupla do número fornecido;
- *void* : utilizado quando o tipo de variável a ser declarada ainda não é conhecido.

Você deve ter reparado que, quando se utilliza o comando *printf*, colocam-se símbolos como *%d* , por exemplo. Faz-se isso porque o compilador precisa saber que tipo de dado ele vai exibir na tela. Veja a tabela abaixo com os principais símbolos:

Tabela 1 - Simbologia de tipo de dados

Símbolo	Tipo de Dado Relacionado
%d	int (inteiro)
%i	int (inteiro)
%f	float (real)
%c	char (caractere)
%e	float ou double
%E	float ou double
%s	string
%x	int (hexadecimal)
%X	int (hexadecimal)
%o	int (octal)
%u	int (unsigned)
%ld	int (long)
%g	float*
%G	float*
%p	ponteiro*

*float**: o compilador avalia a melhor forma de exibir o valor contido na variável, podendo exibí-la utilizando um número do sistema decimal ou em uma equação com o símbolo exponencial.

*ponteiro**: indica o endereço de um ponteiro.

Há também o símbolo especial %n que é utilizado para exibir o numero de caracteres contidos na função printf. Veja:

```
#include <stdio.h>

#include <stdlib.h>

int main () {

        int count;

        printf (“Linguagem%n C\n”, &count);

        printf(“%d”, count); }

        system("pause");

}
```

O programa exibe na tela “Linguagem C 9”.

Vale ressaltar que, para o tipo inteiro em C, há dois modificadores que controlam o tamanho de *bytes* nesse inteiro, são o *short* e o *long*. Essas palavras vêm do inglês e significam, respectivamente, curto e longo. Para utilizá-los, basta colocar a palavra antes do nome int quando for declarar uma variável (você vai aprender como fazer isso mais adiante). Veja o exemplo de declaração da variável n:

Exemplo:

```
short int n
```

Há também os modificadores *signed* e *unsigned*, que vêm também do inglês e podem ser lidas como com e sem sinal, respectivamente. Uma variável do tipo inteiro declarada como *unsigned* armazena apenas

valores positivos. Quando se declara uma variável como inteira, o compilador interpreta-a como sendo *signed*, ou seja, ela pode armazenar valores negativos. Porém, algumas vezes sabe-se que serão armazenados apenas valores positivos na variável, então utiliza-se o modificador *unsigned* da seguinte maneira (para uma variável de nome n):

Exemplo:

unsigned int n

- **Conversão entre tipos**

Algumas vezes é necessário converter uma variável de um tipo para outro, a fim de evitar que o programa apresente resultados.

Por exemplo, suponha que você elabore um programa para calcular o resultado da divisão (n1/n2), sendo n1 e n2 declarados como inteiros. Quando n1 for múltiplo de n2, o programa exibirá o valor exato, caso contrário, apresentará erros.

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

int main ()

{

        int n1, n2;

        float x;

        x= n1/n2;

        printf("x = %f",x);

        system("pause");

}
```

Para n1=20 e n2=3, o programa apresentaria x=6. Para que ele apresente as casas decimais e x=6,666..., reescrevemos o programa da seguinte maneira:

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

int main ()

{

        int n1, n2;

        float x;
```

```
    x= (float)n1/n2;

    printf("x = %f",x);

    system("pause");

}
```

Assim, sempre que for necessário fazer uma conversão, é suficiente colocar o tipo de variável desejado entre parênteses na frente das operações matemáticas que definem o resultado.

▪ Declaração de variáveis

Antes de declarar uma variável, é necessário especificar o seu tipo. Feito isso, existem ainda algumas regras básicas:

1) Não pode conter os seguintes símbolos: "%", "+", "-", "#", "$", "!", entre outros;

2) Pode conter números, desde que não seja iniciado por eles (1a não é permitido, mas a1 é);

3) Pode conter "_" (ex. _a);

4) Não pode ser igual a uma palavra reservada

OBS.: A Linguagem C possui 32 palavras reservadas segundo o padrão ANSI, veja a Tabela 2.

Tabela 2 - Palavras reservadas da Linguagem C segundo o padrão ANSI

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	nion
Const	float	short	unsigned
Continue	for	signed	void
Default	goto	sizeof	volatile
do	if	static	while

Exemplos de declaração de variáveis:

- char NOME[30]; /* armazena um nome que contenha no máximo 30 caracteres*/
- float MEDIA_TURMA; /* armazena a um numero que pode conter varios digitos e casas decimais */
- int IDADE; /* armazena um numero inteiro de poucos dígitos */

Você deve estar se perguntando para que servem os símbolos "/*" e "*/". Eles são usados para que o programador possa fazer comentários no código sem alterá-lo. Tudo que for escrito entre esses dois símbolos não será "lido" pelo compilador quando ele for executar o programa. Comentários são muito importantes para que o programador possa se organizar melhor e para que outras pessoas possam entender o código com mais facilidade.

- **Declaração de constantes**

As constantes são valores que são se alteram durante a execução do programa. Elas podem ser tanto valores inteiros tanto números que contenham várias casas decimais. São declaradas da seguinte forma:

#define nome_da_variavel valor /* nao se utiliza “;” no fim da linha */

Por exemplo:

#define IDADE 20

Operadores

Os operadores são símbolos que, quando associados às variáveis do código, retornam algum valor desejado. Esses operadores podem ser unários, o que significa que só é necessária uma variável para que o operador possa ser aplicado e retornar um resultado. Os operadores são divididos com bases nas suas funções. No fim deste tópico há um código simples que utiliza todos os tipos de operadores. Abaixo estão os grupos de operadores com o respectivo significado de cada operador.

▪ Operadores Aritméticos

Os operadores aritméticos estão relacionados às quatro operações básicas. Os operadores mais simples são os dois operadores unários + e -, que indicam o sinal de dado número. Abaixo está a descrição de cada operador.

Adição e subtração: a adição e a subtração na linguagem C podem ser realizadas com qualquer número de variáveis. Além disso, as duas operações podem ser combinadas entre si para se obter um resultado. Para fazer a adição de duas variáveis 'a' e 'b' basta escrever 'a + b'. Analogamente, para subtrair a variável 'a' de uma variável 'b', basta escrever 'a - b'.

Multiplicação e divisão: a multiplicação e a divisão na linguagem C podem ser efetuadas com a utilização dos operadores aritméticos * e /. Para multiplicar um elemento 'a' pela variável ''b, basta escrever 'a*b'. Analogamente, para realizar a divisão da variável 'a' pela variável 'b' basta digitar 'a/b'. Vale ressaltar que, se as variáveis 'a' e 'b' forem inicializadas como *int* a e *int* b, o quociente da divisão de 'a' por 'b' será a parte inteira do valor real desse quociente. Ou seja, se 'a = 5' e 'b = 2', ao se calcular 'a/b', o resultado mostrado será 2. Caso, contudo, pelo menos uma das variáveis a ou b fosse inicializada como um *float*, ou seja, um real qualquer, o quociente mostrado seria 2.5.

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

        int a,b,c,d;

        printf("Insira os valores de a, b, c e d;\n\n");

        scanf("%i",&a);

        scanf("%i",&b);

        scanf("%i",&c);

        scanf("%i",&d);

        printf("A soma dos quatro valores vale: %i\n\n",a+b+c+d);

        printf("A subtração do terceiro valor pelo quarto valor vale: %i\n\n",c-d);

        printf("O produto dos quatro valores vale: %i\n\n",a*b*c*d);
```

```
	printf("A divisão inteira do primeiro valor pelo segundo vale: %i\n\n",a/b);
	int e = 9;
	float f = 2;
	printf("%f",e/f); //Quando pelo menos um dos valores é float,
	system("pause"); //a divisão não é necessariamente inteira
}
```

- ## Operadores Relacionais e Lógicos

Os operadores lógicos, também conhecidos como operadores comparativos ou relacionais, são chamados dessa forma por serem utilizados quando se deseja avaliar logicamente algum fato que relacione dois valores. Por exemplo, ao se utilizar um *if* é necessário colocar a condição a ser satisfeita para que os comandos dentro desse *if* sejam executados. Essa condição é escrita utilizando-se os operadores lógicos. Abaixo estão listados os operadores e suas funções:

- **x == y:** este operador serve para testar se x é igual à y.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int x,y;
	printf("\nInsira dois valores inteiros:\n");
	scanf("%i",&x);
	scanf("%i",&y);
	if(x == y) {
		printf("\nOs valores sao iguais.\n");
	}
	else {
		printf("\nOs valores sao diferentes.\n");
	}
	system("pause");
}
```

- **x != y:** este operador serve para verificar se x é diferente de y.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int x,y;
	printf("\nInsira dois valores inteiros:\n");
	scanf("%i",&x);
	scanf("%i",&y);
	if(x != y){
		printf("\nOs valores sao diferentes.\n");
	}
	else {
		printf("\nOs valores sao iguais.\n");
	}
	system("pause");
}
```

- **x < y:** este operador testa se x é menor que y.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int x,y;
	printf("\nInsira dois valores inteiros distintos:\n");
	scanf("%i",&x);
	scanf("%i",&y);
	if(x < y) {
```

```
            printf("\nO primeiro valor eh menor que o segundo valor.\n");
        }
        else {
            printf("\nO segundo valor eh menor que o primeiro valor.\n");
        }
        system("pause");
}
```

- **x <= y:** este operador verifica se x é menor ou igual à y.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n, k;
    bool c = true;
    printf("\nInsira o numero de termos do vetor:\n");
    cin>>n;
    int v[n];
    printf("\nInsira os termos do seu vetor\n");
    for(int i = 1; i<=n; i++){      scanf("%i",&v[i]);      }
    printf("\nInsira um valor inteiro.\n");
    scanf("%i",&k);
    for(int i = 1; i<=n; i++) {
        if(v[i]>=k) {
            c = false;
        }
    }
    if(c == true) {
        printf("%i eh uma cota superior para o conjunto das coordenadas do vetor.\n",k);
```

```
        }
        else {
                printf("%i nao eh uma cota superior para o conjunto de coordenadas do vetor.\n",k);
        }
        system("pause");
}
```

- **!x:** este operador verifica se a variável lógica (tipo bool) que nega x é verdadeira ou falsa. Observe que ela será verdadeira se e somente se x for falsa.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int n,k;bool c = false;
        printf("\nInsira o numero de termos do vetor:\n");
        scanf("%i",&n);
        int v[n];
        printf("\nInsira as coordenadas do seu vetor\n");
        for(int i = 1; i<=n; i++) {
                scanf("%i",&v[i]);
        }
        printf("\nInsira um valor inteiro.\n");
        scanf("%i",&k);
        for(int i = 1; i<=n; i++){
                if(v[i]>=k) {
                        c = true;
                }
        }
        if(!c == true)  {
                printf("%i eh cota superior para o conjunto das coordenadas do vetor.\n",k);
```

```
        }
        else {
                printf("%i nao eh cota superior para o conjunto das coordenadas do vetor.\n",k);
        }
        system("pause");
}
```

- **x || y:** este operador informa se a operação ou(or) entre as variáveis lógicas será verdadeira ou falsa. Observe que o resultado retornado é 0 apenas se as duas variáveis tiverem valor false.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        float a, b;
        printf("Insira os extremos para um intervalo fechado:\n");
        scanf("%f",&a);
        scanf("%f",&b);
        float r;
        printf("\nInsira um numero real\n");
        scanf("%f",&r);
        if((a>r)||(b<r)){
                printf("\nEste numero real nao pertence ao intervalo construido.\n");
        }else{
                printf("\nEste numero real pertence ao intervalo construido\n");
        }
        system("pause");
}
```

- **x && y:** este operador avalia se o valor lógico da operação e(and) sobre as variáveis x e y é verdadeiro ou falso. Vale ressaltar que o valor 1 será retornado apenas quando as duas variáveis tiverem valor lógico 1.

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

	float a,b;

	printf("Insira os extremos para um intervalo fechado:\n");

	scanf("%f",&a);

	scanf("%f",&b);

	float r;

	printf("\nInsira um numero real\n");

	scanf("%f",&r);

	if((a<=r)&&(b>=r)){

		printf("\nEste numero real pertence ao intervalo construido.\n");

	}

	else{

		printf("\nEste numero real nao pertence ao intervalo construido\n");

	}

	system("pause");

}
```

Operadores para Manipulação de Bits

A manipulação de *bits* consiste em fazer operações *bit* a *bit* entre dois números binários ou deslocar, mantendo o mesmo número de bits, algum número binário já conhecido. As operações a serem feitas entre os *bits* de número binários podem ser escritas na forma hexadecimal, pois o próprio programa interpretará esses dados e os converterá. Para isso, é necessário saber como escrever um número na forma binária e hexadecimal.

- Forma binária: antes de escrever o número em binário basta colocar 0b.
- Forma hexadecimal: antes de escrever o número em hexadecimal basta escrever 0x.

Depois de inicializadas as variáveis na forma binária ou decimal, são possíveis realizar operações lógicas *bit* a *bit*. As operações são: and(&), or(|), not(~) e xor(^). Os resultados dessas operações seguem a lógica *booleana*. Já o deslocamento de bits, citado anteriormente, pode ser feito para a esquerda ou para a direita. Quando os bits são deslocados para a esquerda, pode-se imaginar que o deslocamento de n bits é o acréscimo de n zeros à direita do número binário. Por exemplo:

$$x = 1110010 \rightarrow x << 3 \rightarrow x = 1110010\underline{000}$$

Os *bits* sublinhados são os novos *bits*. Fazendo as devidas conversões, é fácil ver que essa operação consiste em multiplicar um número decimal por 2^n. No exemplo acima, o número binário 1110010 correspondente à 114 em decimal foi transformado em 1110010000, que corresponde à $912 = 2^3 \times 114$.

O deslocamento para a direita ocorre da seguinte forma:

$$x = 10110 \rightarrow x >> 2 \rightarrow x = \underline{00}101$$

Os bits sublinhados são os novos bits. Vale ressaltar que essa operação corresponde a achar a parte inteira do quociente do número x na forma decimal por dois elevado ao valor do deslocamento. Neste caso, 10110 correspondente à 22 em decimal foi transformado em 00101, que é igual à 5 em decimal, sendo $5 = 22/(2^2)$.

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

        unsigned short a = 0b11010;

        unsigned short b = a<<3;

        printf("%d\n\n",a);

        printf("%d\n\n",b);

        printf("%d\n\n");//O último bit de a é trocado, e o valor decimal tem seu sinal invertido.

        unsigned short c = a|b;//É executada a operação Or bit a bit.

        printf("%d\n\n",c);

        unsigned short d = a&b;//É executada a operação And bit a bit.

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

        unsigned short e = ~a;//É obtido o complemento de 2^16 em relação à ~a, que vale -27.

        printf("%d\n\n",e);

        unsigned short f = a^b;//É executada a operação Xor bit a bit.

        printf("%d\n\n",f);

        system(“pause”);
```

}

- **Operadores de atribuição**

Os operadores de atribuição têm a função de dar algum valor a uma variável inicializada ou mudar o valor de alguma variável existente que já tinha sido definida. O operador de atribuição mais básico é o símbolo de igualdade. Na programação esse símbolo não representa uma equação, mas uma atribuição. Portanto, escrever 'c = 4' no programa significa que a partir de agora o valor de 'c' é 4.

Quando se fala de atribuição por função básica, essa atribuição pode ser de quatro tipos distintos: atribuição por soma, atribuição por subtração, atribuição por multiplicação e atribuição por divisão. A utilidade da atribuição por operação básica e por resto é evitar que o programador digite desnecessariamente. Observe na Tabela 3 que as formas de atribuição que estão lado a lado são equivalentes:

Tabela 3 - Formas de atribuições de variáveis

b = b + a;	b += a;
b = b – a;	b -= a;
b = b*a;	b *= a;
b = b/a;	b /= a;
b = b%a;	b %= a;

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

        int a,b,c,d,e,f,g,h,i,j;

        printf("\nInsira dez valores inteiros.\n");

        printf("a = ");scanf("%i",&a);

        printf("b = ");scanf("%i",&b);

        printf("c = ");scanf("%i",&c);

        printf("d = ");scanf("%i",&d);

        printf("e = ");scanf("%i",&e);

        printf("f = ");scanf("%i",&f);

        printf("g = ");scanf("%i",&g);

        printf("h = ");scanf("%i",&h);

        printf("i = ");scanf("%i",&i);

        printf("j = ");scanf("%i",&j);
```

```
        b += a;

        c -= d;

        e *= f;

        g /= h;

        i %= j;

        printf("\nO comando b += a retorna b = %i",b);

        printf("\nO comando c -= d retorna c = %i",c);

        printf("\nO comando e *= f retorna e = %i",e);

        printf("\nO comando g += h retorna g = %i",g);

        printf("\nO comando i += j retorna i = %i",i);

        system("pause");

}
```

- **Operadores de incremento e decremento**

Esses dois operadores aritméticos servem para, de maneira mais rápida, se obter a soma ou subtração de um número por 1. Assim, se é desejado incrementar a variável a de 1, basta escrever '++a'. Analogamente, para se o usuário desejar decrementar a variável b de 1, basta escrever '--b'.

Vale ressaltar que esse tipo de operador pode ser utilizado de outra maneira. Colocando o operador do outro lado o valor retornado não será o da variável já incrementada (ou decrementada), mas o valor original da variável.

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

        int a,b;

        printf("\nInsira um valor a e um valor b:\n");

        scanf("%d",&a);

        scanf("%d",&b);

        printf("\na++ Mostra o valor de a e depois o atualiza:\na++ = ",a++,"\n");

        printf("a = ",a,"\n");

        printf("++b atualiza e depois mostra o novo valor de b: ",++b);
```

```
    system("pause");
}
```

- **Expressões**

As expressões são combinações de operações que retornam algum valor em C. Uma expressão pode ser composta de variáveis, constantes, operadores ou chamadas de funções. Dessa forma, define-se também o tipo de uma expressão, que é o tipo do valor retornado por ela. Por exemplo:

```
Char p[20];          //p é um vetor de caracteres.
```

Portanto, a expressão

p – 1

é do tipo ponteiro para caractere.

Exemplo:

```
#include<stdio.h>

#include<stdlib.h>

int main(){

    int x[10], *s; // x é um vetor de dez elementos inteiros a serem definidos e s é um ponteiro para inteiro.

    for (int i = 1; i<11; i++){  // Foram definidos os elementos do vetor x.

        x[i] = 1 + i;

    }

    s = x + 3; // Foi definido o ponteiro s como sendo o ponteiro que guarda o endereço do terceiro elemento do vetor x.

    printf("%i",*s); // O resultado retornado é o valor numérico 3.

    printf("%i",s); // O resultado retornado é o endereço de memória do terceiro elemento do vetor x.

    system(“pause”);

}
```

As expressões ou “*subesxpressões*”, que são expressões dentro de expressões compostas podem ser de qualquer tipo, visto que o valor retornado depende apenas de quais operações foram feitas com as variáveis utilizadas.

Comandos de controle de fluxo

As funções de controle de fluxo são funções que determinam as ações de um programa de acordo com uma condição ou parâmetro definido. As principais são as funções de decisão e de repetição que serão melhor explicitadas nos tópicos seguintes. Em sua sintaxe, deve-se utilizar os operadores vistos anteriormente.

- ***if-else***

A função *if-else* realiza um determinado bloco de comandos caso a condição estabelecida seja satisfeita. O bloco executado caso a condição seja satisfeita fica preso ao comando *if*, do contrário, caso a condição falhe, é executado o bloco de comandos contidos em *else*.

Sintaxe:

```
if (condição) {
        bloco de comandos 1
}
else {
        bloco de comandos 2
}
```

Exemplo:

```
if (x<2) {
        printf("x é igual a 2");
}
else {
        printf("x é diferente de 2");
}
```

- ***for***

A função *for* é uma função de repetição, também chamada de função de *loop*, no qual é executado o mesmo bloco de comandos repetidas vezes. A função *for* necessita que seja definido o valor inicial, a condição de parada e o incremento de uma variável que servirá de parâmetro para a função *for*. O *for* é utilizado quando se sabe a quantidade de vezes que o bloco será executado.

Sintaxe:

```
for (condição inicial da variável ; condição de parada ; incremento) {
        bloco de comandos
}
```

Exemplo:

```
for (int i = 0 ; int < 10 ; i++) {
        printf("i = %d\n",i);
}
```

- ***while***

A função *while* também é uma função de repetição na qual se executa um bloco de comandos apenas no caso de sua condição de repetição ser atendida, ou seja, caso a condição de repetição falhe antes da entrada no loop, o bloco não é executado nenhuma vez. A variável utilizada na condição deve ser modificada durante a execução do loop, do contrário, o loop não será encerrado e o programa falhará.

Sintaxe:

```
while (condição) {
        bloco de comandos
}
```

Exemplo:

```
while (x<2) {
        x = x/2;
        printf("x  = %d",x);
}
```

- ***do-while***

Semelhante às funções *for* e *while*, a função do-while também é uma função de repetição, no entanto, diferente do while, o bloco de comandos pertencente a função sempre será executado pelo menos uma vez, pois a condição de repetição apenas é verificada ao final do bloco, diferente da função *while* onde a verificação é feita no início do bloco.

Sintaxe:

```
do {
        bloco de comandos
while (condição);
```

Exemplo:

```
do {
        printf("Digite um número positivo:");
        scanf("%d", &x);
while (x<0);
```

- ***break***

O comando break é utilizado quando se deseja para a execução de um *loop* ou de um bloco de comandos e continuar a execução do programa logo após esse *loop* ou bloco.

Sintaxe

```
break;
```

Exemplo:

```
for (int i = 0 ; int < 10 ; i++) {
        if (i == 5) {
                break;
        }
        printf("x  = %d",x);
}
```

- ***continue***

O comando *continue* é o oposto da função *break*, onde a execução do *loop* é interrompida e reiniciada na próxima interação

Sintaxe

```
break;
```

Exemplo:

```
for (int i = 0 ; int < 10 ; i++) {
        if (i == 5) {
            continue;
        }
        printf("x  = %d",x);
}
```

- ***switch***

A função *switch* é uma função de escolha, onde o bloco de comandos a ser executado será aquele no qual o valor definido é igual ao da variável analisada. Caso nenhuma das opções seja escolhida, o bloco de comandos executado é o da opção *default*.

Sintaxe:

```
switch (variável) {
        case (valor 1):
            bloco de comandos 1;
            break;
        case (valor 2):
            bloco de comandos 2;
            break;
        default:
            bloco de comandos padrão;
}
```

Exemplo

```
switch (x) {
        case (1):
            printf("Opção 1 escolhida\n");
            break;
        case (valor 2):
            printf("Opção 2 escolhida\n");
            break;
        default:
            printf("Opção inválida\n");
```

- ***goto***

O comando *goto* é um comando que salta para um local específico do programa executado, local este determinado por um rótulo que pode estar antes ou depois do *goto*.

Sintaxe :

Nome do Rótulo 1:

goto Nome do Rótulo 1 ou Nome do Rótulo 2;

Nome do Rótulo 2:

Exemplo:

```
Opção:
printf(“Digite um número positivo: ”);
scanf(“%d”, %x);
if (x<0) {
        goto Opção;
}
```

Vetores

Em C, é possível armazenar diversos valores do mesmo tipo em uma mesma variável. Este tipo especial de variável é denominado vetor. Cada valor armazenado neste, será referenciado pelo nome dado ao vetor, sendo diferenciado apenas pelo índice.

Os índices utilizados para referenciar os valores armazenados em um vetor, iniciam sempre do índice zero e terminam no índice indicado na declaração menos um. Para declarar um vetor é utilizado o seguinte formato:

Tipo_da_variável Nome_da_variável[tamanho_do_vetor]

Exemplo:

float vetor[20];

Ao declarar um vetor, o computador reserva um espaço maior de memória, especial para armazenar valores nesta variável. No exemplo acima, o vetor possui vinte espaços "em branco" para armazenar valores nele.

Como dito anteriormente, esses valores serão referenciados ao longo do programa da seguinte forma:

vetor[0], vetor[1], vetor[2],..., vetor[19]

Lembre-se sempre que o compilador não verificará se você está utilizando o índice correto. Se por algum motivo, você indicar um índice que não exista pode vir a ter problemas futuros.

▪ Inicialização pelo programa

Para atribuir valores a um vetor, basta indicar o índice do elemento o qual se deseja armazenar o valor e atribuí-lo ao elemento.

Exemplo:

- int vetor[10];
- vetor[0] = 10;
- vetor[4] = 7;

No exemplo acima, ao elemento de índice 0 está sendo atribuído o valor 10 e ao elemento de índice 4 está sendo atribuído o valor 7.

▪ Inicialização pelo usuário

Para carregar um vetor, podemos fazer como no exemplo anterior, indicar os valores diretamente no código. Ou podemos utilizar um laço *for* para carregar o vetor.

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>
```

```
int main ()
{
        int n;
        printf(“Digite a quantidade de números: ”);
        scanf(“%d”, &n);
        for (i=0;i<n;i++)  {
                printf(“Digite o número de posição %d: ”,i);
                scanf(“%d”, &numero[i]);
        }
        for (i=n-1;i>=0;i--) {
                printf(“Numero de posição %d = %d\n”, i , numero[i]);
        }
        system(“pause”);
}
```

Observe que o laço for foi utilizado para carregar o vetor, variando o seu índice. Em seguida, outro laço for é utilizado para, desta vez, imprimir na tela os valores armazenados no vetor.

- ***getch*()**

A função *getch*() também lê o caractere e retorna para a função que a chamou, porém esta função não permite que o caractere seja impresso na tela, ou seja, recebe o caractere “sem eco”. Veja o exemplo:

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        char letra;
        printf(“Digite a primeira letra do seu nome”);
        letra = getch();
        printf(“A primeira letra do seu nome eh: %c”, letra);
        system(“pause”);
```

}

No exemplo acima, a variável letra recebe o caractere digitado, porém não imprime o resultado na tela, ela apenas armazena o caractere retornado pelo comando *getch*(). Para isso, é necessário colocar a função *printf* logo em seguida. É possível fazer uma analogia com a função *scanf*. Neste caso, seria análogo usar a função scanf ou o comando *getch*(). Porém, é útil lembrar que a função *getch*() não substitui a função *scanf*, pois esta só retorna um caractere.

- ***getche*()**

A função *getche*() lê o caractere que foi digitado e retorna para a função que a chamou. No entanto, esta função exibe na tela o caractere armazenado, ou seja, recebe o caractere "com eco". Veja o exemplo:

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

int main ()

{

        char letra;

        printf("Digite a primeira letra do seu nome");

        letra = getche();

        printf("A primeira letra do seu nome eh: %c", letra);

        system("pause");

}
```

A única diferença entre o programa deste exemplo e o do exemplo anterior, é que neste exemplo, quando a variável letra recebe o valor retornado pelo comando '*getche*()', rapidamente é exibido na tela o caractere armazenado.

Strings

Uma *string* é uma série de caracteres alfanuméricos utilizada não só na linguagem C mas também em outras linguagens de programação. Inicialmente, para se usar uma *string*, é necessário incluir a biblioteca <string.h>.

Para se fazer uma *string*, é utilizado um vetor de caracteres, onde cada posição do vetor representa uma letra ou número. É importante ressaltar que o fim do vetor é identificado pela presença do caractere nulo '\0'. Lembrando também que a primeira posição do vetor é identificada pela posição 0 e não pela posição 1.

- ### Declaração de *strings*

A declaração de uma *string* é feita da seguinte maneira:

char nome_da_string [tamanho];

Como foi dito anteriormente, o fim do vetor é identificado pela presença do caractere nulo '\0', portanto, deve-se reservar um espaço no vetor para esse caractere, ou seja, o vetor deve ser declarado com o tamanho uma unidade maior.

Pode-se também inicializar a *string* no momento da sua declaração sem se preocupar com o tamanho da mesma. Para isso, podem ser utilizadas duas sintaxes diferentes, que são mostradas nos exemplos abaixo:

i) char exemplo[] = "PET Eletrica";

ii) char exemplo[] = {'P', 'E', 'T', ' ', 'E', 'l', 'e', 't', 'r', 'i', 'c', 'a', '\0'};

É importante ressaltar que, nesses casos, a *string* terá sempre o tamanho fixo do número de letras atribuído. Nesse caso, a *string* "exemplo" terá tamanho igual a 12. Repare que, no segundo caso, foi separado um espaço para o caractere nulo, a fim de que seja identificado o fim do vetor.

- ### *Strings* - *printf* e *puts*

Suponha que se deseja colocar na tela uma *string* já definida. Para isso, pode ser utilizada a função *printf*. Para exemplificar, observe o programa abaixo que coloca na tela a frase "Ola, mundo!".

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main()

{

        char frase[] = "Ola, mundo!";

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

```
    system("pause");
}
```

Como se pode ver, a *string* “frase” foi declarada e, simultaneamente, inicializada com a frase “Ola, mundo!”. Em seguida, foi utilizada a sintaxe abaixo para colocá-la na tela:

```
printf(“%s”, nome_da_string);
```

Poder-se-ia também usar a função *puts* para substituir toda essa linha de comando, utilizando a seguinte sintaxe:

```
puts(nome_da_string);
```

- ***Strings - scanf***

Utilizando-se a função *scanf*, é possível que o usuário informe quais os caracteres que serão armazenados na *string*. Para exemplificar, observe o programa mostrado abaixo.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char frase[40];
    printf("Digite a frase desejada: ");
    scanf(“%s”, &frase);
    system("pause");
}
```

Como se pode ver, foi criada uma *string* chamada “frase” com tamanho 40. Em seguida, é utilizada a função *printf* para pedir que o usuário digite a frase desejada e, finalmente, é utilizada a sintaxe abaixo para armazenar os caracteres digitados na *string* “frase”:

```
scanf(“%s”, &nome_da_string);
```

Poder-se-ia também usar a função *gets* para substituir toda essa linha de comando, utilizando a seguinte sintaxe:

```
gets(nome_da_string);
```

- ***Strings - strcpy***

A função *strcpy* é utilizada para copiar o conteúdo de uma *string* para outra. É utilizada a seguinte sintaxe:

strcpy(nome_da_primeira_string,nome_da_segunda_string);

O conteúdo original da primeira *string* é perdido e substituído pelo conteúdo da segunda *string*. É importante lembrar que para que isso ocorra, o tamanho da primeira *string* deve ser, no mínimo, igual ao tamanho da segunda *string*.

Pode-se também utilizar essa função para substituir o conteúdo de uma *string* por uma série de caracteres não inicializada em nenhuma outra *string*, como mostra o exemplo abaixo que substitui o conteúdo da *string* "vetor" pela frase "Ola, mundo!" e coloca a mesma na tela.

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main ()

{

    char vetor[100];

    strcpy (vetor,"Ola, mundo!");

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

    system("pause");

}
```

- ***Strings - strcat* e *strncat***

A função *strcat* é utilizada para concatenar duas *strings*. Sua sintaxe é:

strcat(nome_da_primeira_string,nome_da_segunda_string);

Depois do uso da linha de comando mostrada acima, a primeira *string* conterá os seus caracteres originais concatenados com todos os caracteres da segunda *string*.

Semelhante à função *strcat*, a função *strncat* também é utilizada para concatenar duas *strings*. A diferença é que a função *strncat* concatena apenas uma quantidade, escolhida pelo programador, de caracteres da segunda *string*. Sua sintaxe é:

strncat(nome_da_primeira_string,nome_da_segunda_string,número_de_caracteres);

Exemplo:

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

```
#include <stdlib.h>

#include <string.h>

int main ()

{

        char vetor1[] = “Apostila”;

        char vetor2[] = “deLinguagemC”;

        strcat (vetor1,vetor2);

        printf ("%s\n",vetor1);

        system("pause");

}
```

No exemplo acima, a função *strcat* faz com que a *string* vetor1 contenha o seu conteúdo original concatenado com o conteúdo da *string* vetor2. Ou seja, no final do programa, será mostrada na tela a frase “ApostiladeLinguagemC”.

- ***Strings - strlen***

A função *strlen* é utilizada para retornar o tamanho de uma determinada *string*. Sua sintaxe é mostrada abaixo:

```
strlen(nome_da_string);
```

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main ()

{

        char vetor1[] = "LinguagemC";

        int x;

        x = strlen(vetor1);

        printf ("%d\n",x);

        system("pause");

}
```

No exemplo acima, a função *strlen* é utilizada para armazenar na variável x a distância entre o início da

string e o terminador da *string* vetor1, ou seja, o seu tamanho. No final, o programa mostra na tela o valor de x, ou seja, 10.

- ***Strings - strcmp, strncm, strcmpi* e *strncmpi***

A função *strcmp* é utilizada para verificar se duas *strings* são ou não iguais. Caso elas sejam iguais, essa função retorna zero. No caso da primeira *string* ser maior que a segunda, é retornado o valor -1 e, no caso contrário, é retornado o valor 1, sendo que no alfabeto a menor letra é "a" e a maior é "z". Sua sintaxe é:

strcmp(nome_da_primeira_string,nome_da_segunda_string)

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main ()

{

        char vetor1[] = "LinguagemC";

        char vetor2[] = "aLINGUAGEMC";

        float x;

        x = strcmp(vetor1,vetor2);

        printf ("%f\n",x);

        system("pause");

}
```

No programa acima, a função *strcmp* é utilizada para comparar as *strings* vetor1 e vetor2. É retornado o valor -1 para x, visto que a *string* vetor1 é maior que a *string* vetor2 ("L" > "a"). Esse valor é, então, colocado na tela.

Semelhante à função *strcmp*, a função *strncmp* também é utilizada para comparar duas *strings*. A diferença é que a função *strncmp* compara apenas uma quantidade, escolhida pelo programador, de caracteres das *strings*. Sua sintaxe é:

strncmp(nome_da_primeira_string,nome_da_segunda_string,número_de_caracteres);

Caso não se queira considerar a diferença entre letras maiúsculas e minúsculas, utilizam-se, para os casos anteriores, as funções *strcmpi* e *strncmpi*, respectivamente. As sintaxes utilizadas para essas funções são:

strcmpi(nome_da_primeira_string,nome_da_segunda_string);

strncmpi(nome_da_primeira_string,nome_da_segunda_string,número_de_caracteres);

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main ()

{

        char vetor1[] = "LinguagemC";

        char vetor2[] = “LINGUAGEMC”;

        float x;

        x = strcmpi(vetor1,vetor2);

        printf ("%f\n",x);

        system("pause");

}
```

No programa acima, a função *strcmpi* é utilizada para comparar as *strings* vetor1 e vetor2, sem considerar a diferença entre as letras maiúsculas e minúsculas. Desta forma, é retornado o valor 0, já que as duas *strings* vão ser consideradas iguais, para x. Esse valor é colocado na tela.

Matrizes

Matrizes são estruturas utilizadas para armazenar vários dados de forma simultânea, desde que estes sejam do mesmo tipo. Os vetores, estudados anteriormente, são um tipo de matriz, chamada unidimensional.

- ### Matrizes de *strings*

Essas matrizes caracterizam-se por admitirem variáveis do tipo *char*, ou seja, caracteres. Essa peculiaridade deve ser indicada na declaração da matriz, da seguinte forma:

char nome_da_string [número de linhas][número de colunas]

Uma peculiaridade do tipo *char* é o fato deste reservar o último espaço de cada índice para o caractere '\0', ou seja, o espaço da matriz declarado não será totalmente utilizado com o armazenamento de variáveis.

Observe o exemplo da matriz abaixo, onde o programa lê uma lista de caracteres provenientes do teclado, através da função *gets*(), e as armazena em uma *string* unidimensional:

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main ()

{

        char string_nome[10];

        printf("Digite seu primeiro nome\n");

        gets (string_nome);

        system("pause");

}
```

- ### Matrizes bidimensionais

Vimos que matrizes unidimensionais (Vetores) armazenam todas as variáveis em forma de "linha" na memória. As bidimensionais o fazem em forma de linhas e colunas, cujas dimensões são previamente indicadas em sua declaração, como no exemplo a seguir:

tipo_variável nome_variável [número de linhas][número de colunas];

As variáveis em matrizes são lidas na mesma ordem em que são armazenadas: da esquerda para a direita e de cima para baixo. Ou seja, enquanto uma matriz é percorrida, o índice "número de colunas" varia com maior frequência. Observe o exemplo de como declarar uma matriz.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main ()
{
	int matriz_quadrada [4][4];
	int i, j, variavel;
	printf("Digite 16 valores para variáveis\n");
	for (i=0; i<4; i++) {
		for (j=0; j<4; j++) {
			scanf("%d", &variavel);
			matriz_quadrada[i][j] = variavel;
		}
	}
	for (i=0; i<4; i++) {
		for (j=0; j<4; j++) {
			printf("%d\t", matriz_quadrada[i][j]);
		}
		printf("\n");
	}
	system("pause");
}
```

- **Matrizes multidimensionais**

Também chamada de Matriz de N dimensões, possui a mesma função dos outros tipos apresentados, mas esta adequa-se ao tamanho estabelecido em sua declaração. Funciona basicamente como um bloco de espaços onde se armazenam variáveis. A declaração é feita da seguinte forma:

```
tipo_variável nome_variável [dimensão1][ dimensão2]...[ dimensãoN];
```

Observe o exemplo abaixo, no qual compila-se uma matriz de dimensão quatro, cada uma com dois elementos.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main ()
{
	int matriz_dimensao_quatro [2][2][2][2];
	int i, j, k, l, variavel;
	printf("Digite 16 valores para variáveis\n");
	for (i=0; i<2; i++) {
		for (j=0; j<2; j++) {
			for (k=0; k<2; k++) {
				for (l=0; l<2; l++) {
				scanf("%d", &variavel);
				matriz_dimensao_quatro[i][j][k][l] = variavel;
				}
			}
		}
	}
	printf("\n");
	for (i=0; i<2; i++) {
		for (j=0; j<2; j++) {
			for (k=0; k<2; k++) {
				for (l=0; l<2; l++) {
				printf("%d\t", matriz_dimensao_quatro[i][j][k][l]);
				}
				printf("\n");
			}
			printf("\n");
```

```
            }


        }

        system("pause");

}
```

- **Inicializações**

A inicialização de matrizes é feita a partir de uma lista de valores de mesmo tipo, separados por vírgula ou não (no caso de *strings*), que serão armazenados na forma citada no tópico de matrizes bidimensionais.

Exemplos de inicialização:

- int matriz_quadrada [2][2] = {1, 2, 3, 4};
- char string_nome[10] = “Eletrica”;
- char string_nome[10] = {‘E’, ‘l’, ‘e’, ‘t’, ‘r’, ‘i’, ‘c’, ‘a’, ‘\0’};

É possível inicializar uma matriz sem especificar seu tamanho, este adquire a dimensão da variável fornecida, observe:

char matriz_desconhecida[ ] = {“Tamanho”}

No exemplo, o compilador irá associar o tamanho da variável atribuída à matriz, no caso, a matriz_ desconhecida teria dimensão oito, pois foi inicializada com sete caracteres e o último índice é ocupado pelo ‘\0’.

Ponteiros

Ponteiros são tipos que indicam o endereço de memória ou o conteúdo de certo endereço de memória. Para declarar um ponteiro, é necessário saber o tipo da variável a qual esse ponteiro se refere. Assim, a sintaxe da

declaração de um ponteiro é da seguinte forma:

(Tipo_da_variável) *nome_do_ponteiro;

- **Operadores de ponteiros**

Os operadores de ponteiros são dois: ‘&’ e ‘*’. O primeiro operador, quando acompanhando o nome de uma variável, significa o endereço ao qual aquele ponteiro aponta. O segundo operador, quando à esquerda do nome do ponteiro representa o conteúdo que fica no endereço apontado. O exemplo a seguir deve retornar, primeiro, o endereço da variável ‘a’ e, segundo, o conteúdo da variável ‘a’, que é o próprio valor desta.

Exemplo:

```
#include <stdlib.h>

#include <stdio.h>

int main()

{

        int a = 3;

        int *ponteiro_de_a = &a;

        printf("%p\n",ponteiro_de_a);

        printf("%p\n",*ponteiro_de_a);

        system("pause");

}
```

- **Aritmética de ponteiros**

Os ponteiros admitem poucas operações aritméticas, pois não é possível realizar multiplicação ou divisão com variáveis do tipo ponteiro. As operações possíveis de se realizar entre ponteiro ou em um único ponteiro são: operações de incremento ou decremento, operações relacionais ou lógicas e operações de atribuição.

As operações de atribuição para ponteiros têm sua utilidade óbvia, que é definir o endereço ou o conteúdo de certo ponteiro.

As operações de incremento e decremento podem ser realizadas para se alterar tanto o elemento endereçado pelo ponteiro quanto o endereço para o qual o ponteiro aponta. No primeiro caso, basta utilizar *(nome_do_ponteiro) como variável a ter o valor atribuído, enquanto no segundo caso, a variável a ter valor atribuído é a (nome_do_ponteiro), que mostra o endereço.

Exemplo:

```
#include <stdlib.h>

#include <stdio.h>
```

```
int main()
{
	int a = 3,b=6;
	int *ponteiro_de_a = &a,*ponteiro_de_b=&b;
	printf("%p\n",ponteiro_de_a);
	printf("%i\n",*ponteiro_de_a);
	printf("%p\n",ponteiro_de_b);
	printf("%i\n",*ponteiro_de_b);
	printf("%i\n",*ponteiro_de_a + *ponteiro_de_b);
	system("pause");
}
```

É possível também escrever desigualdades ou outras sentenças de testes lógicos que envolvam ponteiros. Nesse caso, se lida com ponteiros como sendo uma variável comum.

- **Vetores**

Ao se questionar qual a necessidade do uso de ponteiros visto sua semelhança com vetores, pois os vetores também podem ser considerados ponteiros (sendo, porém, constantes) podem-se perceber as principais características que tornam um ponteiro diferente de um vetor. Essas características conferem certas vantagens ao uso de ponteiros. A primeira vantagem está no fato de o ponteiro poder ser incrementado e o vetor não. Além disso, o vetor não pode mudar seu endereço de memória. A segunda vantagem está no fato de o ponteiro não alocar a memória para um conjunto de elementos, como faz o vetor. Isso significa que com um único espaço alocado pelo ponteiro, é possível incrementá-lo de tal forma a obter todos os termos de um vetor.

Exemplo:

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
	int *pont;
	int v[4];
	printf("Insira os elementos do vetor:\n");
	for(int i = 0; i<4; i++) {
		scanf("%i",&v[i]);
```

```
	}
	printf("Mostrando os termos utilizando o conceito de ponteiro:\n\n");
	for(int i = 0; i<4; i++) {
		printf("%i\n\n",*(v+i));
	}
	system("pause");
}
```

- **Funções**

A utilidade prática de se utilizar ponteiros em funções é fazer com que funções sejam argumentos de outras funções. Para isso, basta utilizar um ponteiro que seja do mesmo tipo que a função. Uma função que retorna um inteiro, por exemplo, terá que ter um ponteiro do tipo inteiro para que este possa repassá-la como argumento.

Exemplo:

```
#include <stdlib.h>
#include <stdio.h>
void inserindo(float* valor, float num) {
	for (int i = 0; i < num; i++) {
		printf("Insira o valor %i: ", i + 1,"\n");
		scanf("%f",&valor[i]);
	}
}
void mostrando(float* valor, float num) {
	for (int i = 0; i < num; i++) {
		printf("Valor %i: %f\n",i+1,valor[i]);
	}
}
int main() {
	float v[3];
	inserindo(v, 3);
	mostrando(v, 3);
```

```
        system("pause");
}
```

Caso o programa acima fosse escrito sem utilizar vetores, basta trocar *float** valores por valor[ ].

- ***strings***

Em vários casos uma *string* pode ser mostrada ao usuário por meio do uso de ponteiros. Isso acontece porque se um ponteiro é do tipo *char*, então ele retornará o endereço do primeiro elemento do vetor de caracteres que forma a *string*. Observe o seguinte programa, que utiliza *strings* sem usar ponteiros.

Exemplo:

```
#include <stdlib.h>

#include <stdio.h>

int main() {

        char nome[ ] = "PET Eletrica UFC";

        printf(nome);

        system("pause");

}
```

Agora, utilizando ponteiros, tem-se:

Exemplo:

```
#include <stdlib.h>

#include <stdio.h>

int main() {

        char *nome = " PET Eletrica UFC\n\n";

        printf(nome,"\n\n"); //Retorna a frase: "PET Eletrica UFC".

        printf("%c\n\n",*nome);//Retorna a primeira letra da frase: "P".

        printf("%p\n\n",&nome);//Retorna o endereço do primeiro elemento do ponteiro.

        printf("%c\n\n",*(nome+2));//Retorna o conteúdo do segundo endereço após o endereço do "P".

        system("pause");

}
```

Função

As funções são blocos de comandos utilizados pra otimizar o código, tornando-o mais organizado, evitando que fique grande demais e que o mesmo trecho seja repetido várias vezes, torna mais fácil e prática um possível correção do código e permite que um bloco de comandos seja utilizado posteriormente pelo próprio programador ou por outros usuários.

As funções têm o seguinte formato:

```
tipo_de_retorno nome (lista_de_argumentos)
{
        bloco_de_comandos
}
```

- ### Variáveis locais

São variáveis declaradas dentro de uma função (que é iniciada com "{" e finalizada com "}") e que, fora dessa função, não têm validade. Assim, é possível ter, em um mesmo programa, várias funções, e cada uma delas, com variáveis x, y e z, sem que elas sejam "confundidas" pelo compilador. Variáveis locais são declaradas da mesma forma que as demais.

Exemplo:

```
void funcao1(void) {
        int n;
        n=123;
}
void funcao2(void) {
        int n;
        n=456;
}
```

Note que a variavel n é declarada duas vezes, em funcao1() e em funcao2(), assumindo diferentes valores normalmente.

- ### Variáveis globais

São variáveis declaradas fora de todas as funções do programa. Elas podem ser "chamadas" e modificadas por qualquer função do programa. Sempre que, em uma função, a variável local foi igual a variável global, executar-se-á a variável local. Porém, recomenda-se a utilização de diferentes variáveis para melhor organização do programa, facilitando sua leitura pelo usuário e evitando possíveis confusões para o

programador.

Vale lembrar que as variáveis globais ocupam bastante espaço na memória durante toda a execução do programa, enquanto as locais ocupam somente quando estão sendo usadas. Então, para a otimização do seu programa, procure utilizar as variáveis globais somente quando necessário.

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

float n1,n2; // variaveis declaradas fora da funcao principapal (variaveis globais)

printf("Digite o primeiro numero da soma: " );

scanf("%f", &n1);

printf("Digite o segundo numero da soma: " );

scanf("%f", &n2);

int main() {

        float soma;  // variavel declarada dentro da funcao principal (local)

        soma = n1 + n2;

        printf("%.2f + %.2f = %.2f\n", n1, n2, soma);

        system(“pause”);

}
```

- **void**

A palavra *void* vem do inglês e significa vazio. Na linguagem C, ela indica um tipo de retorno que permite ao programador criar funções que nada retornam, que não possuem parâmetros ou cujo tipo de retorno não é conhecido.

Quando se deseja que a função retorne algo, é necessário utilizar a declaração *return*. Caso não se deseje um retorno da função, usa-se *void*.

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

int square (int a) {

        return (a * a);

}
```

```
void inicio (void) {
        printf ("Oi, seja bem vindo!\n");
}
void fim (void) {
        printf ("Tchau, ate a proxima.\n");
}
int main () {
        int n, result;
        inicio ();
        printf ("Digite um número inteiro: ");
        scanf ("%d", &n);
        result = square (n);
        printf ("O quadrado de %d é %d.\n", n, result);
        fim ();
        return 0;
        system(“pause”);
 }
```

- **Chamado e retorno de uma função**

Para chamar um função, é necessário que ela seja declarada primeiro. Ela pode ser usada em qualquer parte do código, segundo o seguinte modelo:

nome_da_funcao (variavel)

Tome sempre o cuidado de verificar se o tipo de variável é compatível com a função e com o tipo de retorno para evitar possíveis erros.

Para obter o retorno de uma função, é necessário especificar o tipo de retorno antes do nome da função (caso o tipo escolhido for *void*, a função não retornará nada) e utilizar o comando *return*. Caso o tipo de retorno não for especificado, o compilador retornará um inteiro.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int soma(int a, int b)
```

```
{
        int c;
        c = a + b;
        return c;
}
int main()
{
        int n1, n2, x;
        printf("Digite os numeros que voce deseja somar: \n");
        scanf("%d", &n1);
        scanf("%d", &n2);
        x = soma(n1, n2);
        printf("O resultado da soma eh: %d\n", x);
        system(“pause”);
}
```

Note que quando os números a serem somados forem digitados, eles serão armazenados em ‘a’ e ‘b’ e sua soma em ‘c’. O valor armazenado em ‘c’ será o retorno da função acima, uma vez que utilizou-se o comando “*return* c”.

- **Recursividade**

Uma função recursiva é aquela capaz de “chamar” a si mesma. É preciso tomar muito cuidado com funções recursivas para que ela não caia em um *loop* infinito, determinando sempre um critério de parada.

Por exemplo, suponha que você deseja saber o valor da soma de todos os números naturais até 10. É bastante intuitivo pensar em efetuar o cálculo: (10 + 9 + ...+2 +1). Mas se você desejar elaborar um fórmula geral que forneça todos a soma de todos os naturais até n, pode-se aplicar a mesma ideia: (n + (n-1) + (n-2) +...+ 1). Note que foi estabelecido um critério de parada: o último termo da soma deve ser sempre igual a 1. Se esse critério não tivesse sido estabelecido, você poderia cometer o erro de continuar somando mesmo os termos menores que zero, e seu resultado não seria a soma dos naturais até n.

Veja a ideia dessa soma escrita de outra maneira, para que se possa colocar em um código:

$$soma(n) = n + soma(n-1)$$

$$soma(n-1) = (n-1) + soma(n-2)$$

$$soma(n-2) = (n-2) + soma(n-3)$$

...

soma(1) = 1

Perceba que, ao somarmos os dois lados da equação, obtemos:

soma(n) = n + (n-1) + ... +1

A mesma equação obtida anteriormente. Vamos agora escrever um código que efetue essa conta, usando a ideia de recursividade:

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

int soma(int n)

{

	if(n == 1) {

		return 1;

	}

	else {

		return (n + soma(n-1));

	}

}

int main()

{

	int n, x;

	printf("Digite um numero natural: ");

	scanf("%d", &n);

	x = soma(n);

	printf("A soma de todos os naturais ate %d eh: %d\n",n, x);

	system("pause");

}
```

Perceba que foram utilizados dois blocos de funções, um para efetuar a soma e outro para ler o valor de entrada e imprimir o resultado. Note também que foi estabelecido um critério de parada:

```
if(n == 1) {
```

```
    return 1;
}
```

Isso garantiu que a função não entrasse em um *loop* infinito.

Struct

Uma *struct* é um tipo de variável que armazena diversas variáveis em uma só, podendo estas serem de diferentes tipos. Fazendo uma breve analogia, a *struct* funciona como uma ficha de inscrição, onde a pessoa deve preencher com o seu nome, telefone e endereço (três informações de diferentes tipos). Desta forma, a *struct* é útil para armazenar variáveis de diferentes tipos em uma variável de um tipo só.

- ### Declarando uma *struct*

A declaração de uma *struct* segue o seguinte padrão:

```
struct nome_da_estrutura {

        tipo_da_variavel nome_da_variavel_1;

        tipo_da_variavel nome_da_variavel_2;

        tipo_da_variavel nome_da_variavel_3 ;

        ...

        tipo_da_variavel nome_da_variavel_n;

};
```

Acima, foi definido um tipo de variável *struct*. Neste tipo de variável, contém outras variáveis de diversos tipos. No exemplo abaixo, você entenderá melhor:

Exemplo:

```
struct ficha_de_inscricao {

        char nome[30], endereco[50];

        int telefone;

};
```

No exemplo acima, foi criado um tipo de variável denominado ficha_de_inscricao. Variáveis do tipo ficha_de_inscricao, armazenarão informações em três variáveis distintas, são elas: nome (variável do tipo *char*, que armazena até 30 caracteres), endereço (variável do tipo *char* que armazena até 50 caracteres) e telefone (variável do tipo *int*).

- ### Acesso as variaveis da *struct*

Depois de definida a *struct*, é possível definir variáveis daquele tipo e ao longo do programa manipular os valores armazenados. Veja o exemplo a seguir:

```
#include <stdlib.h>

#include <stdio.h>
```

```
#include <string.h>

struct ficha_de_inscricao {

        char nome[30], endereco[50];

        int telefone;

};

main(void)

{

        struct ficha_de_inscricao ficha; // Está definindo a variável 'ficha' do tipo 'ficha_de_inscrição'

        strcpy(ficha.nome, "Joao da Silva"); // Está armazenando a string'João da silva' no vetor 'nome' armazenado na variável 'ficha'

        strcpy(ficha.endereco, "Rua Jose das Flores"); // Está armazenando a string' Rua José das Flores' no vetor 'endereco' armazenado na variável 'ficha'

        ficha.telefone = 99887766;

        printf("Nome: %s\nEndereco: %s\nTelefone: %d\n", ficha.nome, ficha.endereco, ficha.telefone);

        system("pause");

}
```

Repare que se a variável ‘ficha’ fosse um vetor, seria possível armazenar diversas fichas de inscrição na mesma variável. Esse é um artifício muito utilizado para armazenar diversas informações em uma variável.

- ***malloc***

A função *malloc*() serve para alocar uma determinada quantidade de *bytes* consecutivos e retorna um ponteiro apontando para o “endereço” do primeiro *byte* alocado. O número de *bytes* alocados deve ser definido no argumento da função. Veja a sintaxe:

```
void *malloc(size_t numero_de_bytes);
```

Você entenderá melhor a utilização da função *malloc* no próximo tópico (*sizeof*), onde será implementado um código utilizando os dois comandos.

- ***sizeof***

Muitas vezes não se sabe o tamanho da variável que se vai trabalhar. Para isso, serve o operador *sizeof*. Este operador retorna o tamanho da variável utilizada e sua sintaxe é:

```
sizeof variável;
```

Neste caso, *sizeof* está determinando o tamanho da variável ‘variavel’.

Uma ferramenta muito utilizada é unir a função *malloc*() ao operador *sizeof*.

valor = (int*) malloc (sizeof variavel);

Muitas vezes, não se sabe o tamanho da variável que se vai trabalhar. Aliando a função *malloc* ao *sizeof* este problema é resolvido.

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>

int main(){

	int n,i,*p,sum=0;

	printf("Determine a quantidade de valores a serem somados: ");

	scanf("%d",&n);

	p=(int*) malloc(n*sizeof(int));

	if(p==NULL) {

		printf("Error! memory not allocated.");

		exit(0);

	}

	printf("De entrada com os elementos do vetor: ");

	for(i=0;i<n;++i) {

		scanf("%d",p+i);

		sum+=*(p+i);

	}

	printf("Soma=%d",sum);

	free(p);

	return 0;

	system(“pause”);

}
```

No exemplo acima, a função *malloc* alocará um bloco de memória suficiente para alocar ‘n’ valores do tipo inteiro e retornará um ponteiro apontando para o primeiro valor alocado. Desta forma, o ponteiro ‘p’ assume um papel de “vetor”, sendo utilizado para armazenar valores em *bytes* de memória.

- ***union***

Uma declaração *union* reserva uma única posição de memória para alocar diversas variáveis diferentes. Veja sua sintaxe:

```
union identificador{
        tipo1 nome_da_variavel1
        tipo2 nome_da_variavel2
        tipo3 nome_da_variavel3
        ...
} variáveis_union;
```

Para entender melhor, veja o programa a seguir:

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
union trabalho {
        float bolsa;
        long int matricula;
}u;
int main(){
        printf("Digite sua matricula:\n");
        scanf("%d",&u.matricula);
        printf("Digite o valor da sua bolsa: \n");
        scanf("%f",&u.bolsa);
        printf("Display:\nMatricula :%d\n",u.matricula);
        printf("Salario: %f\n",u.bolsa);
        system("pause");
}
```

No programa acima, primeiramente foi declarado uma variável ‘u’, onde foram armazenados no local de memória desta, duas outras variáveis: ‘bolsa’ do tipo ‘*float’* e ‘matricula’ do tipo ‘*int’*. Em seguida, estas variáveis são implementadas, de modo que ao serem citadas, devem ser da forma: variável_union.nome_da_variavel.

Exemplo: u.bolsa e u.matricula.

- ***enum***

Ao declarar um tipo de variável como *enum*, o compilador entenderá que toda variável daquele tipo só poderá assumir os valores determinados na declaração. Para isto, o compilador associa a cada variável declarada um valor inteiro, conforme a sua ordem (No exemplo abaixo: segunda = 0, março =2 ...). Veja a sua sintaxe:

enum identificador {lista_de_enumeracao} lista_de_variaveis;

Veja o programa exemplo a seguir:

```
#include <stdio.h>

#include <stdlib.h>

enum ano{ janeiro, fevereiro, marco, abril, maio, junho, julho, agosto, setembro, outubro, novembro, dezembro};

int main(){

        enum ano mes;

        mes=agosto;

        printf("%do mes\n",mes+1);

        system("pause");

        return 0;

}
```

No programa acima, é possível ver claramente como funciona a lógica do comando enum. A saída do programa acima será: “8o mês”.

- ***typedef***

O comando *typedef* possibilita ao programador alterar o nome de um tipo de variável. Veja a sintaxe do comando:

typedef tipo novo_nome

O comando typedef é muito útil para declarar tipos de variável struct. Veja o exemplo a seguir.

Exemplo:

```
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

typedef struct ficha_de_inscricao {

        char nome[30], endereco[50];
```

```
        int telefone;
} ficha;
main(void) {
        ficha ficha1; // Está definindo a variável 'ficha' do tipo 'ficha_de_inscrição'
        strcpy(ficha1.nome, "Joao da Silva"); // Está armazenando a string'João da silva' no vetor 'nome' armazenado na variável 'ficha'
        strcpy(ficha1.endereco, "Rua Jose das Flores"); // Está armazenando a string' Rua José das Flores' no vetor 'endereco' armazenado na variável 'ficha'
        ficha1.telefone = 99887766;
        printf("Nome: %s\nEndereco: %s\nTelefone: %d\n", ficha1.nome, ficha1.endereco, ficha1.telefone);
        system("pause");
}
```

O exemplo acima, já foi apresentado anteriormente. Porém, desta vez está sendo utilizado o comando *typedef* para alterar o tipo de variável “struct ficha_de_inscricao” para “ficha”.

Arquivos

A gravação em arquivo é a maneira mais eficiente de se armazenar uma grande quantidade de dados, pois este o faz de forma permanente, gravando em disco, ou seja, estas informações podem ser acessadas mesmo com o encerramento do programa em que foi criado tanto por este, como por demais programas.

Existem duas formas de arquivos em C, destinados diferentes tipos de dados, sãos estes:

- **Arquivos de texto:** são aqueles que armazenam informações do tipo caractere, que podem ser exibidos em tela durante a execução do programa ou acessados, e até modificados, em um editor de texto, durante ou posteriormente ao fechamento do programa. Esta forma de arquivo é sequencial, ou seja, é necessária sua leitura completa mesmo para buscar qualquer informação nele contida.

- **Arquivos binários:** são aqueles formados por uma sequencia de *bits* que seguem a estrutura estabelecida no programa que os criou. Diferente do arquivo de texto, as informações em binário podem ser acessadas em qualquer posição, sem a necessidade da leitura completa do mesmo, o que caracteriza um arquivo randômico ou de acesso aleatório.

Como citado, os arquivos podem ser acessados através de várias fontes, para tal, a Linguagem C possui uma série de comandos específicos para manipulações de arquivos, independente de seu conteúdo. Para que seja possível a aplicação destas funções, os arquivos devem ser declarados no tipo ponteiro (*FILE*), da seguinte forma:

```
FILE *ponteiro_arq;
```

fopen

Esta é a função responsável pela abertura de arquivos, permitindo sua manipulação. Para isso, é necessário informar o nome do arquivo e o modo de abertura.

```
FILE *ponteiro_arq;

ponteiro_arq = fopen ("nome_arquivo.tipo","modo");
```

O tipo citado acima é representado por txt (arquivo de texto) ou dat (arquivo binário).

O modo de abertura pode ser de três tipos:

- **r**: abre um arquivo já existente no disco, somente para leitura;

- **w**: cria um arquivo para gravar informações. Caso exista arquivo com o nome indicado, este será formatado, tendo todo o seu conteúdo anterior excluído para receber os novos dados;

- **a**: cria ou edita um arquivo para gravação. Caso exista um arquivo com o nome indicado, seu conteúdo será conservado, apenas acrescentando-se as novas informações ao final deste.

Veja um exemplo de abertura de arquivo no modo "a":

```
#include <stdlib.h>

#include <stdio.h>
```

```
#include <windows.h>

int main()

{

        FILE *ponteiro_arq;

        ponteiro_arq = fopen (“exemplo.txt”,”a”);

        if (ponteiro_arq == NULL) {

                printf (“Erro de criação de arquivo”);

        }

        else {

                printf(“Seu arquivo foi criado com sucesso!”);

        }

        fclose(ponteiro_arq);

        printf(“pause”);

}
```

Vale ressaltar que um arquivo não pode ser acessado por mais de um programa simultaneamente, a segunda execução não será efetivada, retornando um erro ao usuário. Outro erro de abertura pode ocorrer quando o arquivo a ser acessado para leitura estiver bloqueado.

A função *fclose* utilizada ao final do código é a responsável pelo fechamento do arquivo, para que este seja salvo.

- ***putc***

Esta é uma função de escrita em arquivos, responsável por gravar caracteres informados pelo usuário no arquivo declarado. Lembrando que este caractere deve ser gravado em uma variável declarada anteriormente.

putc (nome_da_variável, ponteiro_do_arquivo);

Veja o exemplo a seguir:

```
#include <stdlib.h>

#include <stdio.h>

#include <windows.h>

int main()

{

        FILE *ponteiro_arq;
```

```
        char variavel;
        ponteiro_arq = fopen (“exemplo.txt”,”a”);
        printf(“Digite o caractere a ser gravado em arquivo:”);
        scanf (“%c”, &variavel);
        putc (variável, ponteiro_arq);
        fclose(ponteiro_arq);
system(“pause”);
}
```

- ***getc***

É a função responsável pela leitura de um arquivo caractere por caractere, geralmente utilizado dentro de um laço de repetição, para que todo o arquivo seja percorrido pela função e tenha seus caracteres impressos na tela. É definida da seguinte forma:

```
caractere = getc (ponteiro_do_arquivo);
```

Exemplo:

```
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>
int main() {
        FILE *ponteiro_arq;
        char caractere;
        ponteiro_arq = fopen ("exemplo.tipo","a");
        if (ponteiro_arq == NULL) {
                printf ("Erro de criação de arquivo");
        }
        else {
                do{
                        caractere = getc(ponteiro_arq);
                        printf("%c", caractere);
                }while (caractere=! EOF);
```

```
        }
        system("pause");
}
```

Programas em Linguagem C

- ### Programa Dev-C/C++

O Dev-C++ é um ambiente de desenvolvimento integrado (IDE – Integrated Development Enviroment) para linguagens C e C++, ou seja, é um *software* que facilita a tarefa de programação em linguagem C e C++ não só por conter um editor de texto com diversos recursos mas também por possuir um compilador. O Dev-C++ possui versões tanto para Windows como para Linux. Na Fig. 1 é mostrada a sua tela principal.

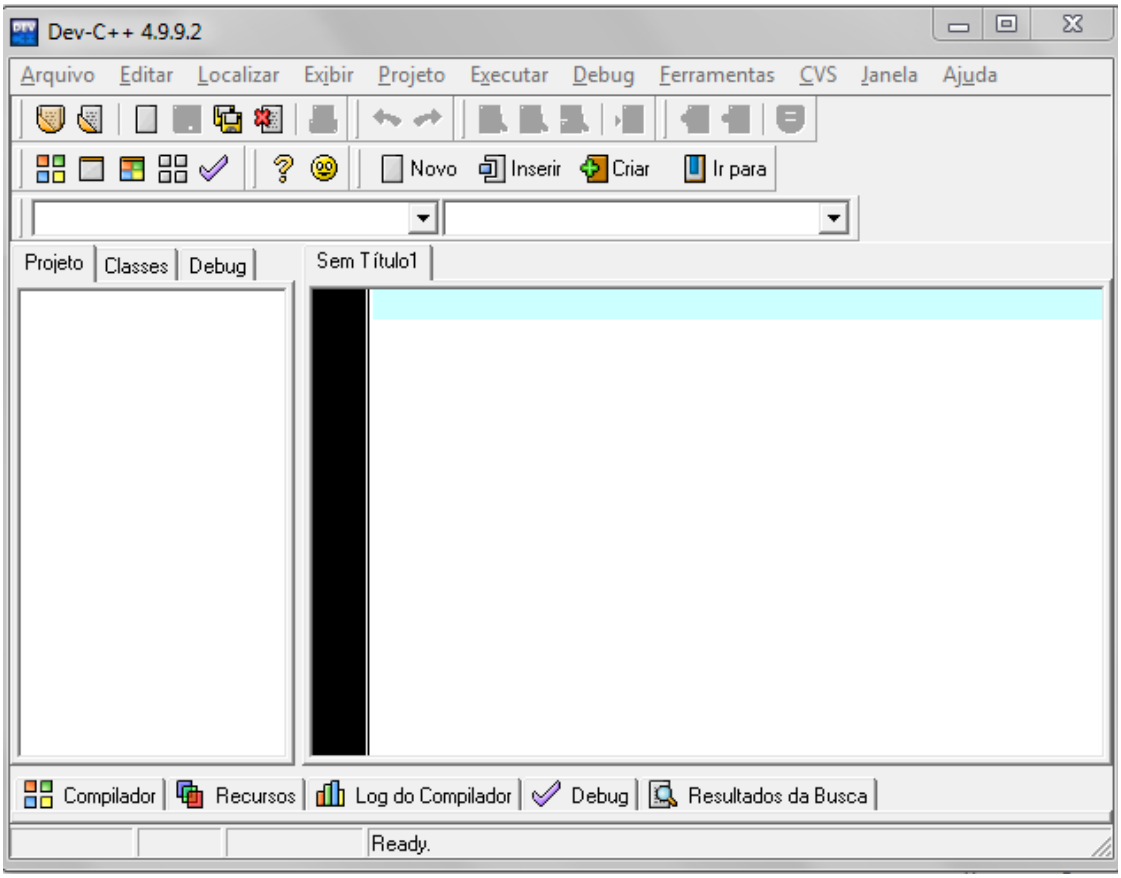


Figura 1 - Tela principal do Dev–C ++.

- A barra de tarefas principal

A barra de tarefas principal contém os comandos mais utilizados no Dev-C++ (estes comandos também podem ser acessados pelo menu ou por atalhos no teclado).

Figura 2 - Barra de tarefas principal.

- **Abrir Projeto ou Arquivo (Ctrl+O):** Abre um arquivo ou projeto anteriormente gravado. Podem ser abertos mais de um arquivo. Cada arquivo é aberto em uma nova aba.

- **Arquivo fonte (Ctrl+N):** Cria um novo arquivo fonte em uma nova aba onde é possível escrever

um algoritmo de programação em linguagem C.

- **Salvar (Ctrl+S):** Grava o texto presente na aba que está em uso. Na primeira vez que um novo texto é gravado, o Dev-C++ pede seu nome e sua localização.
- **Salvar Todos:** Salva o texto presente em todas as abas.
- **Fechar (Ctrl+F4):** Fecha a aba que está em uso.
- **Imprimir (Ctrl+P):** Imprime na impressora padrão o texto presente no editor.

Várias dessas funções também podem ser acessadas do menu Arquivo que é mostrado na Fig. 3.

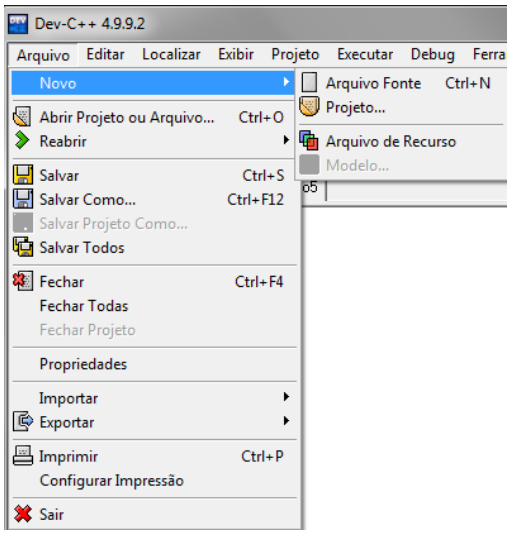


Figura 3 - Menu Arquivo.

- O Menu de Comandos

O Dev-C++ possui um menu de comandos com 11 opções, mostrado na Fig. 4, que possibilitem executar diversas tarefas operacionais. Você poderá ter acesso a esse menu de três formas diferentes:

i) Pressione a tecla de função <F10> e, em seguida, utilize as teclas setas para movimentar o cursor sobre as opções desejadas.

ii) Pressione a tecla <ALT> juntamente com a letra que estiver grifada em maiúsculo, que é a primeira letra de cada opção do menu.

iii) Utilizando o mouse, dê um clique sobre a opção desejada.

Para sair do menu de qualquer caixa de diálogo que venha a ser acionada, basta pressionar a tecla <ESC>.

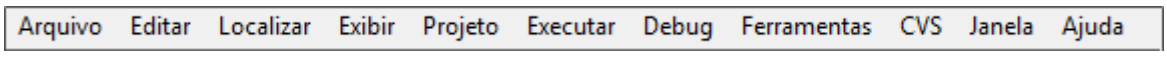


Fig. 4 - Menu de comandos.

- Arquivo

Esta opção possibilita executar operações básicas de controle com os arquivos. É possível: criar um novo

arquivo (Novo), abrir um programa existente (Abrir), salvar um programa em disco (Salvar), salvar um programa em disco com outro nome (Salvar Como), salvar todas as abas (Salvar Todos), fechar a aba ativa (Fechar), fechar todas as abas (Fechar Todas), imprimir o arquivo da aba ativa (Imprimir) e sair do programa (Sair).

- Editar

Esta opção possibilita executar operações de editor do programa, sendo possível remover, movimentar e copiar vários textos que estejam selecionados. É possível: desfazer (Desfazer) e refazer (Refazer) operações efetuadas com a edição, Remover o texto previamente selecionado (Cortar), copiar um texto selecionado do editor para uma área de transferência (Copiar), copiar um texto da área de transferência para o editor (Colar), selecionar todo o texto pertencente ao editor (Selecionar Todos), comentar trechos do programa (Comentar) e descomentar trechos do programa (Descomentar), criar marcas de acesso rápido para partes do programa (Criar Bookmarks) e acessar marcas de acesso rápido (Ir para Bookmarks).

- Localizar

Esta opção possibilita executar comandos de procura e substituição de partes do código. É possível: localizar uma sequencia de caracteres (Localizar), substituir uma sequencia de caracteres por outra (Substituir) e mover o cursor para uma linha previamente selecionada (Ir para Linha).

- Exibir

Esta opção permite o controle de quais componentes da tela são exibidos.

- Projeto

Esta opção refere-se a projetos de programas que possuem vários componentes e arquivos de códigos separados. É utilizado para adicionar e retirar componentes do projeto.

- Executar

Esta opção possibilita executar os comandos básicos do compilador. É possível: compilar o programa da aba ativa (Compilar), executar o programa da aba ativa (Executar), compilar e executar o programa da aba ativa (Compilar & Executar) e procurar por erros de sintaxe (Checar Sintaxe).

- Debug

Esta opção serve para controlar o *debug* de um programa, que é a sua execução passo-a-passo para melhor análise e busca por erros.

- Ferramentas

Esta opção refere-se a várias opções do compilador, do ambiente de trabalho e de edição, além de outras diversas configurações.

- CVS

Esta opção é uma função extra do compilador.

- Janela

Esta opção possui comandos úteis para quando há vários arquivos abertos ao mesmo tempo. É possível: fechar todos os arquivos abertos (Fechar todas), entrar no modo tela cheia (Tela Cheia), ir para próxima aba aberta (Próxima) ou ir para aba anterior (Anterior) e selecionar a aba que se deseja editar (Lista).

- Ajuda

Esta opção dá acesso à ajuda do Dev-C++, que possui uma listagem dos principais comandos do compilador e um breve tutorial da linguagem C.

Compilador de Dev-C/C++

Ver tópico "Criando Layouts", acho que já tem lá

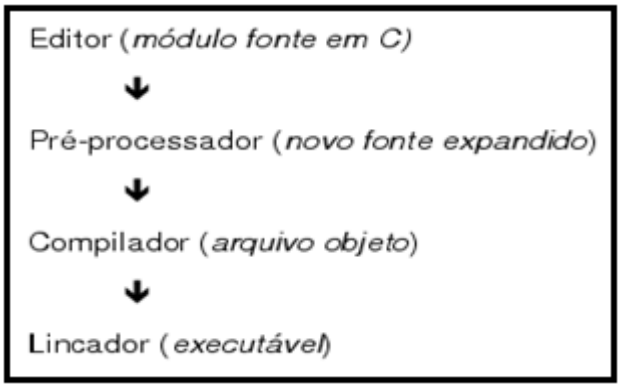


Figura 5 - Processos até a criação de um programa executável.

Os computadores utilizam internamente o sistema binário. Através deste sistema, todas as quantidades e todos os valores de quaisquer variáveis poderão ser expressos através de uma determinada combinação de dígitos binários, ou seja, usando apenas os algarismos 1 e 0. O computador necessita que alguém ou algo traduza as informações colocadas no código fonte (aquele escrito pelo programador em uma determinada linguagem) para um código escrito apenas com 1 e 0. Este código escrito com o sistema binário é chamado de **código executável**.

O programa responsável por converter um código-fonte em programa executável (binário) é o **compilador**. Ao processo de conversão denominamos de **compilação**.

O tempo em que o código é transformado de código fonte escrito em uma linguagem de programação para o código em linguagem de máquina (código objeto) é denominado **tempo de compilação**. O tempo em que o programa está sendo executado é denominado **tempo de execução**.

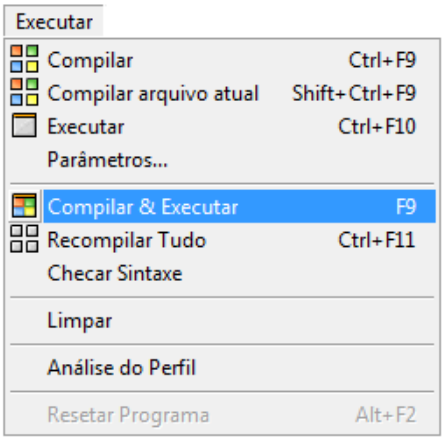


Figura 6 - Compilar e executar um programa no DEV-C++.

Para se compilar e criar um programa executável no DEV-C++, depois de escrito o código fonte, deve-se, primeiramente, clicar em Executar e, em seguida, clicar em Compilar & Executar, como mostra a Fig. 6. Outra forma de se fazer isso é apertando-se a tecla <F9> no teclado.

Feito isso, será aberta uma tela igual à mostrada na Fig. 7 para que seja salvo o seu arquivo .c. Digite o nome que você deseja para o arquivo e clique em Salvar.

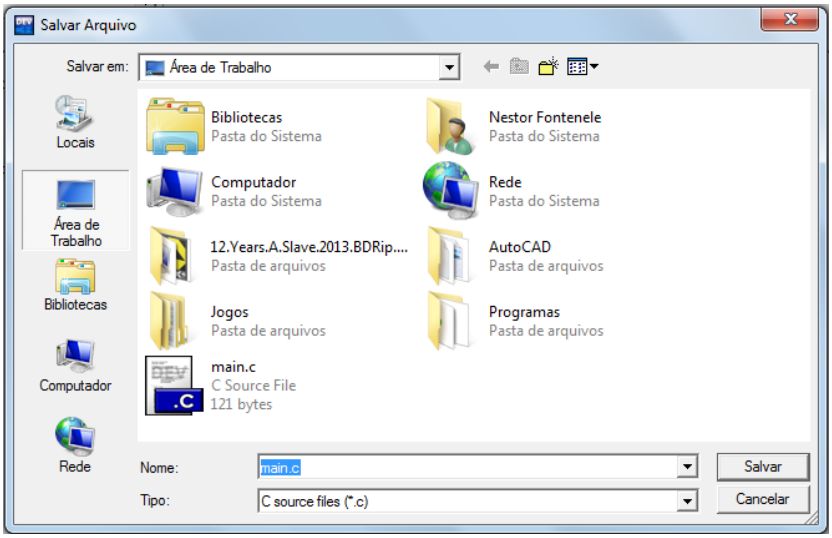


Figura 7 - Salvar arquivo .c.

Depois de salvo o seu arquivo .c, será aberta uma tela igual à mostrada na Fig. 8 que mostrará o progresso do processo de compilação do seu arquivo fonte. Caso não haja nenhum erro, o seu programa executável será criado com sucesso.

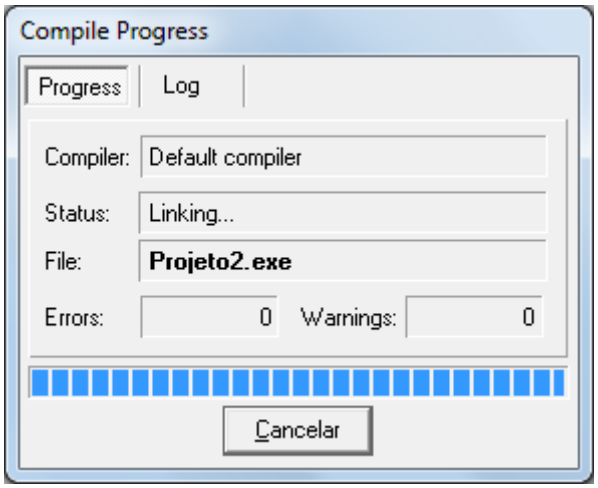


Figura 8 - *Compile Progress.*

- **Programa exemplo**

Para exemplificar a tela que é mostrada depois que o seu programa é compilado e executado, digite no editor de texto do DEV-C++ o código mostrado abaixo que, como se pode perceber, tem como função apenas colocar na tela o valor da divisão de 20 por 5.

Exemplo:

```
#include <stdio.h>

#include <stdlib.h>


int main()

{

	int x=20,z;

	z = x/5;

	printf("O valor de %d/5 eh %d\n",x,z);

	system(“pause”);

	return 0;

}
```

Realize os passos ensinados no tópico anterior para compilar e executar o seu programa. Caso você tenha feito tudo corretamente, uma tela como a mostrada na Fig. 9 deverá ser aberta. Como era de se esperar, ele coloca na tela a frase “O valor de 20/5 eh 4” e salta uma linha devido ao comando “\n”. Observe que, logo depois disso, aparece na tela a frase “Pressione qualquer tecla para continuar...”. No caso de ser apertada alguma tecla, essa tela deverá ser fechada.

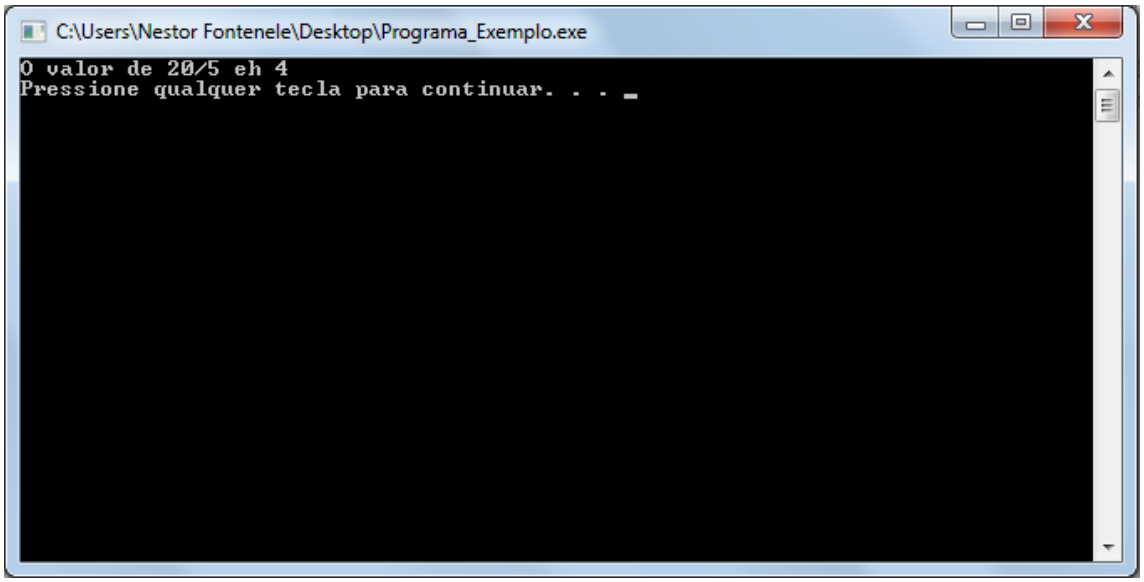


Figura 9 - Programa Exemplo.

- **Programa executável**

Como já citado anteriormente, depois de compilado o seu código fonte, é criado, então, um arquivo .c, como mostrado na Fig. 10. Ao clicar duas vezes sobre esse arquivo, o código fonte do mesmo é aberto no *software* DEV-CE++, precisando ser executado para que seja mostrada, por exemplo, a tela da Fig. 9.

Figura 10 – Arquivo .c.

Além de ser criado o arquivo .c que contém o código fonte digitado pelo programador, após compilar esse código fonte, também é criado um arquivo .exe, como mostrado na Fig. 11. Ao clicar duas vezes sobre esse arquivo, o programa executável deverá ser aberto, sem precisar do compilador do DEV-C++.

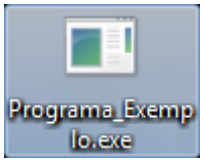


Figura 11 – Arquivo .exe.

Lista de comandos

- ***system***

Comando utilizado para executar tarefas diretamente no sistema do computador. Os comandos utilizados deverão estar entre aspas.

Sintaxe:

system(“comando”);

Exemplo:

system(“pause”); //aguarda o usuario apertar uma tecla para encerrar a execução do programa

- ***color***

Comando utilizado para alterar tanto as cores do plano de fundo e dos textos do programa. O código das cores é: 0 = Preto, 1 = Azul, 2 = Verde, 3 = Verde-água, 4 = Vermelho, 5 = Roxo, 6 = Amarelo, 7 = Branco, 8 = Cinza, 9 = Azul claro, 10 = Verde claro, 11 = Verde-água claro, 12 = Vermelho claro, 13 = Lilás, 14 = Amarelo claro, 15 = Branco brilhante. Este comando deve ser utilizado no comando *system*.

Sintaxe:

system (“ color numéro da cor do fundo e do texto”);

Exemplo:

system(“color 14”); //coloca a cor no fundo preto e a letra azul

- ***cls***

Comando utilizado para limpar a tela de execução do programa. Este comando deve ser utilizado no comando *system*.

Sintaxe:

system(“cls”);

- ***textbackground***

Comando utilizado para alterar apenas a cor de fundo do programa. O código de cores é o mesmo do comando *color*.

Sintaxe:

textbackground(número da cor);

Exemplo:

textbackground(1); //deixará o fundo com cor azul

- ***textcolor***

Comando utilizado para alterar apenas a cor do texto. O código de cores é o mesmo do comando *color*.

Sintaxe:

textcolor(número da cor);

Exemplo:

textcolor(4); // deixara o texto com a cor vermelha

- ***clrscr***

Comando utilizado para limpar a tela de execução do programa, semelhante ao comando system("cls").

Sintaxe:

clrscr;

- ***gotoxy***

Comando utilizado para posicionar o texto na tela numa posição dada.

Sintaxe:

Gotoxy (número da coluna, número da linha)

Exemplo:

gotoxy(6,7); //Deixará o texto na coluna 6 e na linha 7

- ***Sleep***

Comando utilizado para retardar a execução do programa em milisegundos.

Sintaxe:

Sleep(tempo em milisegundos);

Exemplo:

Sleep(600); //Deixa o programa 600 milisegundos mais lento naquele dado momento onde este comando é executado.

- ***return***

Comando utilizado para retornar um valor a uma dada função.

Sintaxe:

```
tipo_de_saída function(variáveis_de_entrada) {
        comandos;
        return(variável_de_saída);  // a função function irá retornar o valor zero
}
```

Exemplo:

```
int quadrado (int a) {
        n = a*a;
        return(n);
}
```

- ***exit***

Comando utilizado para retornar um valor a uma dada função e encerrar o programa imediatamente.

Sintaxe:

```
tipo_de_saída function(variáveis_de_entrada) {
        comandos;
        exit(variável_de_saída);  // a função function irá retornar o valor zero
}
```

Exemplo:

```
int quadrado (int a) {
        n = a*a;
        printf("%d ao quadrado eh %d",a,n);
        exit(n);
}
```

- ***abort***

Comando utilizado para para a execução do prgrama em um determinado ponto Retorna um valor de erro padrão do sistema.

Sintaxe:

```
abort();
```

- ***fopen***

Comando utilizado para abrir um arquivo (*txt ou *dat) no programa.

Sintaxe:

FILE *nome_do_ponteiro

nome_do_ponteiro = fopen (“nome_do_arquivo”,“modo”)

Exemplo:

FILE *ponteiro_arq;

ponteiro_arq = fopen (“exemplo.txt”,“a”);

- ***fclose***

Comando utilizado para fechar um arquivo (*txt ou *dat) no programa.

Sintaxe:

fclose (nome_do_ponteiro)

Exemplo:

FILE *ponteiro_arq;

ponteiro_arq = fopen (“exemplo.txt”,“a”);

fclose (ponteiro_arq)

- ***fgetc***

Comando utilizado para ler um caractere de um arquivo (*txt ou *dat). Após a leittura, faz com que oponteiro aponte para o próximo caractere.

Sintaxe:

fgetc(nome_do_ponteiro);

Exemplo:

char caractere;

FILE *ponteiro_arq;

caractere = fgetc(ponteiro_arq);

- ***fgets***

Comando utilizado para ler uma *string* de um arquivo (*txt ou *dat).

Sintaxe:

fgets(string, numero_de_caracteres+1, nome_do_ponteiro);

Exemplo:

char texto [100];

FILE *ponteiro_arq;

fgets(texto, 100, ponteiro_arq); // o 'fgets' lê até 99 caracteres ou até o '\n'

- ***fputc***

Comando utilizado para enviar um caracter para um arquivo (*txt e *dat).

Sintaxe:

fputc(variavel, nome_do_ponteiro);

Exemplo:

char caractere = ‘P’;

FILE *ponteiro_arq;

fputc(caractere, ponteiro_arq);

- ***fputs***

Comando utilizado para enviar uma *string* para um arquivo (*txt e *dat).

Sintaxe:

fputs(string, nome_do_ponteiro);

Exemplo:

char texto[10] = “PET”;

FILE *ponteiro_arq;

fputs(texto, ponteiro_arq);

- ***fread***

Comando utilizado para ler dados de um arquivo (*txt e *dat).

Sintaxe:

fread(endereço_da_variável, tamanho_lido, quantiade, nome_do_ponteiro);

Exemplo:

FILE *ponteiro_arq;

texto[5];

fread(&texto,sizeof(texto),1,ponteiro_arq);

- ***fwrite***

Comando utilizado para escrever dados em um arquivo (*txt e *dat).

Sintaxe:

fwrite(endereço_da_variável, tamanho_lido, quantiade, nome_do_ponteiro);

Exemplo:

FILE *ponteiro_arq;

texto[5] = “PET”;

fwrite (&texto,sizeof(texto),1,ponteiro_arq);

- ***getchar***

Comando utilizado para ler um caractere do teclado e retornar um número inteiro. É a função padrão de leitura de caractere.

Sintaxe:

getchar();

Exemplo:

int c

c = getchar();

- ***getch***

Comando utilizado para ler um caractere do teclado sem eco, ou seja, o caractere digitado não aparece na tela, e armazená-lo em uma variável.

Sintaxe:

getch();

Exemplo:

char c;

c = getch();

- ***getche***

Comando utilizado para ler um caractere do teclado com eco, ou seja, o caractere digitado aparece na tela, e armazená-lo em uma variável.

Sintaxe:

getch();

Exemplo:

char c;

c = getch();

- ***putchar***

Comando utilizado para escrever o caractere corresponde a um valor inteiro em ASCII.

Sintaxe:

putchar(variavel);

Exemplo:

int caractere = 80;

putchar(caractere); //imprime na tela a letra 'P' que em código ASCII é representado pelo valor 80 em decimal

- ***getc***

Comando utilizado para ler um caractere do teclado.

Sintaxe:

getc();

Exemplo:

char caractere;

caractere = getc();

- ***gets***

Comando utilizado para ler uma *string*.

Sintaxe:

fgets(string);

Exemplo:

char texto [100];

fgets(texto);

- ***putc***

Comando utilizado para enviar um caracter à tela do programa.

Sintaxe:

putc(variavel);

Exemplo:

char caractere = ‘P’;

putc(caractere);

- ***puts***

Comando utilizado para escrever uma *string* na tela do programa.

Sintaxe:

puts(string);

Exemplo:

char texto[10] = “PET”;

puts(texto);

- ***scanf***

Comando utilizado para armazenar textos ou valores digitados pelo usuário em uma variável.

Sintaxe:

scanf(“tipo de dados”, &variavel);

Exemplo

scanf(“%d”, &numero);

- ***printf***

Comando utilizado para imprimir textos ou valores na tela de execução do programa.

Sintaxe:

print(“texto”); ou print(“tipo_de_dados”, variavel);

Exemplo:

char texto[20] = “Hello word!”;

printf(“A mensagem que sera: impressa eh: %s”, texto);

- ***sin***

Comando utilizado para calcular o valor do seno do valor X. O valor X deve estar em radianos.

Sintaxe:

sin(x);

Exemplo:

float pi=3.14, y = sin(pi);

printf("O seno de pi eh %f",y);

- ***cos***

Comando utilizado para calcular o valor do cosseno do valor X. O valor X deve estar em radianos.

Sintaxe:

cos(x);

Exemplo:

float pi=3.14, y = cos(pi);

printf("O cosseno de pi4 eh %f",y);

- ***tan***

Comando utilizado para calcular o valor da tangente do valor X. O valor X deve estar em radianos.

Sintaxe:

tan(x);

Exemplo:

float pi=3.14, y = tan(pi);

printf("A tangente de pi eh %f",y);

- ***asin***

Comando utilizado para calcular o valor do arco seno do valor X. O valor retornado estará em radianos.

Sintaxe:

asin(x);

Exemplo:

float a=0, y = asin(a);

printf("O arco seno de 0 eh %f",y);

- ***acos***

Comando utilizado para calcular o valor do arco cosseno do valor X. O valor retornado estará em radianos.

Sintaxe:

acos(x);

Exemplo:

float a=0, y = acos(a);

printf("O arco cosseno de 0 eh %f",y);

- ***atan***

Comando utilizado para calcular o valor do arco tangente do valor X. O valor retornado estará em radianos.

Sintaxe:

atan(x);

Exemplo:

float a=0, y = atan(a);

printf("O arco tangente de 0 eh %f",y);

- ***sinh***

Comando utilizado para calcular o valor do seno hiperbólico do valor X. O valor X deve estar em radianos.

Sintaxe:

sinh(x);

Exemplo:

float pi=3.14, y = sinh(pi);

printf("O seno hiperbólico de pi eh %f",y);

- ***cosh***

Comando utilizado para calcular o valor do cosseno hiperbólico do valor X. O valor X deve estar em radianos.

Sintaxe:

cosh(x);

Exemplo:

float pi=3.14, y = cosh(pi);

printf("O cosseno hiperbólico de pi eh %f",y);

- ***tanh***

Comando utilizado para calcular o valor da tangente hiperbólica do valor X. O valor X deve estar em radianos.

Sintaxe:

tanh(x);

Exemplo:

float pi=3.14, y = tanh(pi);

printf("A tangente hiperbólica de pi eh %f",y);

- ***exp***

Comando utilizado para calcular o valor de 'e' elevado a um exoente.

Sintaxe:

exp(valor);

Exemplo:

y = exp(2);

printf("e^2 = %f",y);

- ***log***

Comando utilizado para calcular o valor do logaritmo natural de x.

Sintaxe:

log(x);

Exemplo:

y = log(exp(2));

printf("log(e^2) = %f",y);

- ***log10***

Comando utilizado para calcular o valor do logaritmo na base 10 de x.

Sintaxe:

log10(x);

Exemplo:

y = log10(100);

printf("log10(100) = %f",y);

- ***modf***

Comando utilizado para separar o número x em inteiro e em fração. A parte inteira será armazenada em y e retorna a fração.

Sintaxe:

modf(x,y);

Exemplo:

z = modf(2.5,y);

- ***pow***

Comando utilizado para calcular o valor x elevado ao valor y.

Sintaxe: pow (x,y);

Exemplo:

x = pow(2,3);

- ***sqrt***

Comando utilizado para calcular a raiz quadrada do valor x.

Sintaxe:

sqrt (x);

Exemplo:

x = sqrt(16);

- ***ceil***

Comando utilizado para arredondar o valor x para o maior inteiro mais próximo.

Sintaxe:

ceil(x);

Exemplo:

x = ceil(2.4);

- ***floor***

Comando utilizado para arredondar o valor x para o menor inteiro mais próximo.

Sintaxe:

floor(x);

Exemplo:

x = ceil(2.6);

- ***rand***

Comando utilizado para gerar um número aleatório entre 0 e *RAND_MAX*, constante definida na biblioteca <*stdlib.h*>.

Sintaxe;

rand();

Exemplo:

x = rand();

- ***srand***

Comando utilizado para diversificar os valores gerados pelo comando *rand*.

Sintaxe:

srand();

Exemplo:

srand((unsigned)time(NULL));

- ***fabs***

Comando utilizado para calcular o valor absoluto de x.

Sintaxe:

fabs(x);

Exemplo:

x = fabs(-5.1);

- ***mod***

Comando utilizado para calcular o valor do resto da divisão de x por y. Equivalente ao operador %.

mod(x, y);

Exemplo:

x = mod(8, 3) ou x = 8%3

- ***strlen***

Comando utilizado para calcular o comprimento de uma string.

Sintaxe:

strlen(string);

Exemplo:

x = strlen(“PET”);

- ***strcpy***

Comando utilizado para copiar o conteúdo de uma *string* para outra.

Sintaxe:

strcpy (string_destino,string_origem);

Exemplo:

char texto[5];

strcpy (texto, “PET”);

- ***strncpy***

Comando utilizado para copiar um determinado número de caracteres de uma *string* para outra.

Sintaxe:

strncpy (string_destino, string_origem, n);

Exemplo:

char texto[5];

strncpy (texto, “PET”,1);

- ***strcat***

Comando utilizado para concatenar, juntar, duas *strings*.

Sintaxe:

strcat(string_destino,string_origem);

Exemplo:

texto[15] = “PET ”;

strcat (texto, “Elétrica”);

printf(“%s”,texto);

- ***strncat***

Comando utilizado para concatenar, juntar, uma determinada quantidade de uma string a outra string.

Sintaxe: strncat (string_destino,string_origem,n);

Exemplo:

texto[30] = “PET ”;

strncat (texto, “Engenharia Elétrica”, 12);

printf(“%s”,texto);

- ***strcmp***

Comando utilizado para comparar duas strings.

Sintaxe:

strcmp(string1,string2);

Exemplo:

x = strcmp (“PET-EE”, “pet-ee”);

- ***strchr***

Comando utilizado para localizar a primeira ocorrência de um caractere em uma string.

Sintaxe:

strchr(string, ‘caractere’);

Exemplo:

x = strchr (“Engenharia”, ‘a’);

- ***strrchr***

Comando utilizado para localizar a última ocorrência de um caractere em uma string.

Sintaxe:

strrchr(string, ‘caractere’);

Exemplo:

x = strrchr (“Engenharia”, ‘a’);

- ***strrev***

Comando utilizado para inverter o conteúdo de uma string.

Sintaxe:

strrev(string);

Exemplo:

texto[10] = “EE-TEP”

strrev(texto);

printf(“%s”,texto);

- ***strstr***

Comando utilizado para localizar uma string dentro de outra string.

Sintaxe:

strstr(string-mae, string-filha);

Exemplo:

x = strstr (“Engenharia Elétrica”, “El”);

- ***strupr***

Comando utilizado para converter as letras minúsculas de uma string em maiúsculas.

Sintaxe:

strupr(string);

Exemplo:

texto[20] = “Engenharia Elétrica”;

strupr(texto);

printf(“%s”,texto);

- ***strlwr***

Comando utilizado para converter as letras maiúsculas de uma string em minúsculas.

Sintaxe:

strlwr(string);

Exemplo:

texto[20] = “Engenharia Elétrica”;

strlwr(texto);

printf(“%s”,texto);

Exercícios

1. Faça um programa que receba do usuário um valor inteiro e atribua a variável 'x', atribua o triplo desse valor a variável 'y' e imprima na tela os valores de 'x' e 'y'.

2. Faça um programa que receba do usuário um valor em Reais e imprima na tela o valor em Dólares e em Libras Esterlinas.
   DICA 1: Considere 1 Dólar = R$ 2,00 e 1 Libra Esterlina = R$ 4,00

3. Faça um programa que receba do usuário o nome, o peso e a altura de uma pessoa e imprima na tela o seu IMC, a sua situação corporal e quanto deve perder ou ganhar, em Kg, para chegar ao Peso Ideal (IMC Ideal = 21,75).
   DICA 1: IMC = $\frac{Peso\ (Kg)}{Altura^2\ (m^2)}$
   DICA 2:

IMC - Situação Corporal	
Abaixo de 17	Muito abaixo do peso
Entre 17 e 18,49	Abaixo do peso
Entre 18,5 e 24,99	Peso normal
Entre 25 e 29,99	Acima do peso
Entre 30 e 34,99	Obesidade I
Entre 35 e 39,99	Obesidade II (severa)
Acima de 40	Obesidade III (mórbida)

4. Faça um programa que receba do usuário um número inteiro de segundos e imprima a quantidade correspondente em Horas, Minutos e Segundos no formato HH:MM:SS.

5. Faça um programa que receba do usuário três valores e indique se eles podem formar ou não um triângulo. Caso possa, indique também se é um triângulo 'ESCALENO', 'ISÓSCELES', ou 'EQUILÁTERO'.

6. Faça um programa que receba do usuário o valor de três provas. Pergunte ao usuário se realizou a prova substitutiva:
   - Se NÃO, calcule a média do aluno;
   - Se SIM, entre com a nota da prova, substitua a menor nota dentre as três iniciais e calcule a nova média.

7. Faça um programa que receba do usuário o valor de quatro notas, calcule a média e imprima na tela a sua média e a sua situação:
   - Se Média menor do que 4 - REPROVADO;
   - Se Média maior ou igual a 4 e menor do que 7 - AF;
   - Se Média maior ou igual a 7 - APROVADO POR MÉDIA.

   Caso a situação seja AF, receba uma quinta nota, calcule a média final entre a primeira média e a quinta nota e imprima na tela sua média final e sua nova situação:
   - Se Média Final menor do que 5 - REPROVADO;
   - Se Média final maior ou igual a 5 - APROVADO.

8. Faça um programa que receba do usuário o valor de um produto, a quantidade de unidades compradas desse produto e imprima na tela o valor da compra à vista, com desconto de 10%, e o valor à prazo, sem desconto.

9. Faça um programa que receba do usuário o valor de temperatura , pergunte ao usuário a 'unidade' da temperatura:

- c para graus Celsius (ºC);
- f para Fahrenheit (ºF);
- k para graus kelvin (K).

Imprima na tela o valor da temperatura nas três unidades (ºC, ºF e K).

10. Faça um programa que receba do usuário os coeficientes a, b e c de uma equação de segundo grau e imprima na tela as raízes da equação nas três situações:

- Raízes reais e diferentes;
- Raízes reais e iguais;
- Raízes complexas.

11. Faça um programa que receba do usuário um número e indique o mês correspondentes. Caso o número seja inválido, peça um novo número ao usuário.

12. Faça um programa que receba do usuário um par ordenado (X,Y) e indique o quadrante ou o eixo a que pertence o par e suas coordenadas polares (módulo e argumento, em graus).

DICA 1: Módulo = $\sqrt{x^2 + y^2)}$

DICA 2: Argumento = $\tan^{-1}\left(\frac{y}{x}\right)$

13. Faça um programa que receba do usuário um número inteiro positivo e calcule o seu fatorial.
DICA 1: Fatorial de N = N! = N*(N-1)*(N-2)*...*2*1
DICA 2: Fatorial de 0 = 0! = 1

14. Faça um programa que receba do usuário a idade, a massa e o sexo de dez pessoas. Calcule e imprima:

- Total de homens;
- Total de mulheres;
- Média das idades dos homens;
- Média das massas das mulheres.

15. Faça um programa que receba do usuário uma letra e indique se é ou não uma VOGAL.

16. Faça um programa que receba do usuário o tempo, em segundos, de 10 voltas de um piloto em uma corrida de Fórmula 1, imprima na tela o valor das 10 voltas em Minutos e Segundos (MM:SS) e indique a volta e o tempo da pior volta, da melhor volta e o tempo médio das 10 voltas.

17. Faça um programa que receba do usuário um numero na base decimal e converta para a base binária e para a base hexadecimal.
DICA 1: Base binária é a base em que os termos são escritos apenas em 0 e 1. O binário é escrito usando o resto da divisão de um número, em decimal, por 2 e alocando os números de trás para frente. Exemplo: 20 em decimal é 10100 em binário. 10 em decimal é 1010 em base binária.
DICA 2: Base decimal é a base em que os termos são escritos entre 0 e 9.
DICA 3: Base hexadecimal é a base em que os termos são escritos entre 0 e F, onde A=10, B=11, C=12, D=13, E=14 e F=15. O hexadecimal é escrito usando o resto da divisão de um número, em decimal, por 16 e alocando os números de trás para frente. Exemplo: 21 em decimal é 15 em hexadecimal. 31 em decimal é 1F em hexadecimal.

18. Faça um programa que receba do usuário um número inteiro e indique se o número é PAR ou ÍMPAR.

19. Faça um programa que receba do usuário um número inteiro e imprima na tela todos os seus divisores positivos.

20. Faça um programa que receba do usuário dois números inteiros e imprima na tela o MMC (Mínimo Múltiplo Comum) e o MDC (Máximo Divisor Comum) entre os dois números.

21. Faça um programa que receba do usuário um número inteiro e imprima na tela se o número é PRIMO ou não.
DICA 1: O Número 1 NÃO é primo.

22. Faça um programa que imprima na tela os N primeiros números PRIMOS.

23. Faça um programa que imprima na tela os N primeiros QUADRADOS PERFEITOS.

24. Faça um programa que imprima na tela os N primeiros termos da SEQUÊNCIA DE FIBONACCI: 1 1 2 3 5 8 13 21 34...

25. Faça um programa que receba do usuário dois vetores de 5 termos, imprima na tela os termos dos dois vetores em ordem crescente e as seguintes operações:

- A União B
- A intersecção B;
- A complementar em B;
- B complementar em A.

DICA 1: Termos de um vetor X complementar em Y representam os termos existentes em X que não existem em Y.

26. Faça um programa que receba do usuário dois vetores de 5 termos e imprima na tela as seguintes operações: Soma termo a termo; Subtração termo a termo; Produto termo a termo; Produto interno destes dois vetores.
DICA 1: Produto interno de A e B = A(1)*B(1) + A(2)*B(2) + ... + A(N)*B(N)

27. Faça um programa que receba do usuário uma frase de até 50 caracteres, incluindo o espaço, e imprima na tela a quantidade de cada vogal ('a', 'e', 'i', 'o' e 'u') e de espaços em branco.
DICA 1: Digite apenas letras minúsculas, pois a Linguagem C é *case sensitive*, ou seja 'A' é diferente de 'a'.

28. Faça um programa que receba do usuário uma frase de até 100 caracteres e imprima na tela a quantidade de vogais, de espaços em branco e de consoantes.

29. Faça um programa que receba do usuário uma frase de até 100 caracteres, armazene as vogais em um vetor, as consoantes em outro e imprima na tela os dois vetores.

30. Faça um programa que receba do usuário uma frase de até 100 caracteres e retorne a quantidade de palavras da frase.

31. Faça um programa que receba do usuário uma frase de até 100 caracteres e retorne a quantidade de vezes que um caractere qualquer aparece na frase.

32. Faça um programa que receba do usuário o nome de 10 pessoas e imprima-os na tela em ordem alfabética.

33. Faça um programa que receba do usuário uma senha de 6 caracteres. Limpe a tela em seguida e peça novamente ao usuário a senha. A digitação das duas senhas deve aparecer sem eco, ou seja, ao invés de surgirem as letras na tela, devem surgir asteriscos ('*').

- Caso o usuário erre após três tentativas, imprima na tela a mensagem 'CONTA BLOQUEADA, ENTRE EM CONTATO COM O SUPORTE'.
- Caso o usuário acerte, a senha surge na tela com a mensagem 'SENHA CORRETA. BEM-VINDO PROGRAMADOR!'

34. Faça um programa que receba do usuário dois caracteres de 'a' a 'z' e imprima na tela a quantidade de caracteres entre eles no alfabeto.
DICA 1: Caso o usuário não digite os caracteres em ordem alfabética, peça os dois caracteres novamente.

35. Faça um programa que receba do usuário o nome de duas pessoas e imprima na tela a quantidade de letras existentes em cada um.

- Caso o nome das pessoas sejam iguais, imprima na tela "AS DUAS PESSOAS SÃO HOMÔNIMAS";
- Do contrário, imprima na tela "AS DUAS PESSOAS SÃO HETERÔNIMO".

36. Faça um programa que receba uma palavra e indique se é ou não um PALÍNDROMO.
DICA 1: Palavras palíndromas são palavras que mantes seu sentido quando lidas de trás para frente.

37. Faça um programa que receba do usuário o nome completo de uma pessoa e retorne a abreviatura deste nome.
DICA 1: Não se devem abreviar as preposições como: do, de, etc.
DICA 2: A abreviatura deve vir separada por pontos. Ex: Paulo Jose de Almeida Prado. Abreviatura: P.J.A.P

38. Faça um programa que receba duas palavras e verifique se uma delas é *substring* da outra, ou seja, se uma palavra está contida na outra.

39. Faça um programa que receba uma frase de no máximo 100 caracteres e retorne a quantidade de encontros vocálicos existentes na frase

40. Faça um programa que receba duas *strings* de no máximo 50 posições, indique qual das duas *strings* tem maior tamanho e os seus respectivos tamanhos. Caso as *strings* tenham o mesmo tamanho, verifique se elas são iguais. Agrupe as duas *string* em uma só.

41. Faça uma função que calcule o valor aproximado do arco tangente de X em radianos, onde X pertence ao intervalo fechado [0,1], através da série $\tan^{-1}(X) = X - \frac{X^3}{3} + \frac{X^5}{5} - \frac{X^7}{7}$ .... Imprima o valor de X em radianos e o seu correspondente em Graus.

DICA 1: A série só deve ser encerrada quando o módulo de $\frac{X^K}{K}$ for menor que 10^{-4}.

DICA 2: $\pi = 180^o$

42. Faça uma função que receba do usuário um valor inteiro n, onde N>=1 e retorne o valor da série $S = \frac{2}{4} + \frac{5}{5} + \frac{10}{6} + \cdots + \frac{n^2+1}{n+3}$

43. Faça uma função que receba do usuário dois valores, A e B, e retorne o valor de A^B. Não utilizar funções ou operadores de potência prontos, como o *'pow'*.

44. Faça uma função que imprima na tela os N primeiros termos primos acima de 100.

45. Faça um programa que receba do usuário a ordem de duas matrizes (m_A x n_A e m_B x n_B), receba os termos de cada matriz, indique se os produtos AxB e BxA podem ocorrer e imprima na tela o resultado de cada produto. Caso um dos produtos não possa ocorrer imprima uma mensagem na tela indicando qual é este produto.

46. Faça um programa que receba do usuário a ordem e os termos de uma matriz e retorne o maior e o menor termo e suas posições.

47. Faça um programa que receba do usuário a ordem e os termos de uma matriz e coloque os termos de cada linha em ordem crescente. Imprima a nova matriz na tela.

48. Faça um programa que receba do usuário uma matriz 5x5 e troque a 2ª linha com a 5ª linha e coloque os termos destas linhas em ordem decrescente. Imprima a nova matriz na tela.

49. Faça um programa que receba do usuário a ordem e os termos de uma matriz e imprima na tela:

- a soma dos termos de cada linha;
- a soma dos termos de cada coluna.

50. Faça um programa que receba do usuário a ordem e os termos de uma matriz e imprima na tela: a matriz; a matriz transposta; os produtos AxA^t e A^txA .

51. Faça um programa que receba do usuário a ordem e os termos de uma matriz quadrada e:

- imprima na tela os termos da diagonal principal;
- imprima na tela os termos da diagonal secundária;
- imprima na tela as duas matrizes com os termos das duas diagonais trocados entre si.

DICA 1: Para uma matriz 3x3, A_{11} troca com A_{13}.

52. Faça um programa que receba do usuário a ordem e os termos de uma matriz quadrada e calcule o traço e o produto dos termos da diagonal principal desta matriz.

53. Faça um programa que receba do usuário a ordem e os termos de uma matriz e um valor qualquer. Imprima a matriz original e a matriz multiplicada pelo valor de entrada.

54. Faça um programa que receba do usuário os termos de uma matriz 4x4 e calcule o seu determinante.

55. Faça um programa que receba do usuário os termos de uma matriz 3x3 e calcule a sua inversa.

56. Faça um programa que receba do usuário um vetor e calcule a matriz de *Vandermonde* a partir deste vetor.

57. Faça um programa que gere uma matriz tridimensional 5x5x5, com valores randômicos entre 5 e 15, e imprima na tela.
DICA 1: Imprima as quantidade de submatrizes da terceira dimensão uma embaixo da outra, destacando o número da submatriz.

58. Faça um programa que declare quatro ponteiros e mostre o valor dos quatro ponteiros e seus respectivos endereços, sendo que o quarto ponteiro é a soma dos 3 primeiros.

59. Faça um programa que declare seis ponteiros e mostre o valor dos seis ponteiros e seus respectivos endereços, sendo que o sexto ponteiro é a média dos cinco primeiros.

60. Faça um programa que receba do usuário um vetor de 10 números entre 0 e 9 e retorne o número de ocorrências de um determinado número. Utilize ponteiros.

61. Faça um programa que receba do usuário um vetor de 10 números entre 0 e 9, inverte a ordem dos números deste vetor e imprime na tela a nova ordem. Utiliza ponteiros.

62. Faça um programa com a estrutura aluno, onde os parâmetros de entrada são o nome completo, a idade e a matrícula do aluno.

63. Faça um programa com a estrutura gado com um total de 100 componentes com os seguintes parâmetros:

- codigo: código da cabeça de gado;
- leite: número de litros de leite produzido por semana;
- alim: quantidade de alimento ingerida por semana - em Kg;
- nasc: data de nascimento - mês e ano (O Campo nascimento também é uma estrutura com os subcampos Mês e Ano.);
- abate: 'N' (não) ou 'S' (sim) - O campo abate é definido como 'S' (sim) no caso de:
    - Ter mais de 5 anos, ou;
    - Produzir menos de 40 litros de leite por semana, ou;
    - Produzir entre 50 e 70 litros de leite por semana e ingira mais de 50 Kg de alimento por dia.

Imprima na tela as seguintes informações:

- Quantidade total de leite produzido por semana;
- Quantidade total de alimento consumido por semana;
- Quantidade total de leite que será produzido por semana após o abate;
- Quantidade total de alimento que será consumido por semana após o abate;
- Quantidade total de cabeças de gado que irão para o abate.

DICA 1: Utilize valores randômicos inteiros para preencher os campos de idade, leite produzido e alimento ingerido. Atenção aos limites mínimos e máximos dos valores randômicos.

64. Faça um programa com a estrutura supermercado que tenha como parâmetros:

- nome (até 20 caracteres);
- setor (A, B, C, D ou E);
- quantidade (inteiro);
- preço por unidade (real de duas casas decimais).

Crie um menu com as seguintes opções:

- Cadastrar um novo produto;
- Mostrar os produtos já cadastrados;
- Mostrar os produtos de determinado setor;
- Calcule o capital investido por setor e no total do estoque;
- Sair do programa.

DICA 1: Para cada nova execução do menu, limpe a tela. O menu deve sempre ficar a mostra.

65. Faça um programa que crie um arquivo de texto, no qual seja armazenado o seu nome completo, idade e curso. Em seguida, acesse-o na pasta em que foi salvo através de um editor de texto.

66. Faça um programa que leia o arquivo criado no exercício anterior e o modifique, acrescentando informações como seu endereço e a data de hoje.

67. Faça um programa que crie arquivo de texto, com as informações do item anterior atualizadas. Em seguida, faça a leitura do arquivo, imprimindo todas as informações na tela.