

Faculdade de Engenharia de Sorocaba

ALGORITMOS E PROGRAMAÇÃO

Conceitos e Exemplos

Engenharia de Computação
Versão 1.1

Profª Talita Berbel

Sorocaba
2019



Sumário

1. INTRODUÇÃO.....	3
2. ESTRUTURA INICIAL DE UM PROGRAMA	6
3. VARIÁVEIS E CONSTANTES.....	10
3.1. VARIÁVEIS.....	10
3.2. CONSTANTES	12
4. COMANDOS DE ENTRADA E SAÍDA	14
4.1. FUNÇÃO PRINTF()	14
4.2. FUNÇÃO SCANF().....	15
5. OPERAÇÕES	18
5.1. OPERADOR DE ATRIBUIÇÃO.....	18
5.2. OPERADORES ARITMÉTICOS	18
5.3. OPERADORES RELACIONAIS.....	21
5.4. OPERADORES LÓGICOS	22
6. ESTRUTURA DE DECISÃO – IF	23
6.1. COMANDO IF	23
6.2. COMANDO IF ELSE	24
6.3. COMANDO IF ELSE ANINHADOS	25
7. ESTRUTURA DE SELEÇÃO – SWITCH	27
8. ESTRUTURA DE REPETIÇÃO – FOR	30
9. ESTRUTURA DE REPETIÇÃO – WHILE e DO-WHILE	36
9.1. REPETIÇÃO WHILE.....	36
9.2. REPETIÇÃO DO-WHILE	39
10. VETORES.....	42
11. MATRIZES.....	46
APÊNDICE A: Acentuação na Linguagem C	50

1. INTRODUÇÃO

Antes de iniciar os primeiros estudos relacionados a programação é necessário fixar alguns conceitos fundamentais, os quais são: lógica de programação, algoritmo e programa.

O primeiro consiste na maneira correta, ordenada e não ambígua de desenvolver uma tarefa, ação ou procedimento, e é aplicado juntamente com conceitos de programação, por esta razão denomina-se lógica de programação.

A **lógica de programação** é, portanto, uma técnica de encadear pensamentos para se atingir um objetivo, ou seja, consiste em compreender claramente os diversos passos e funções que são realizados na execução de um programa.

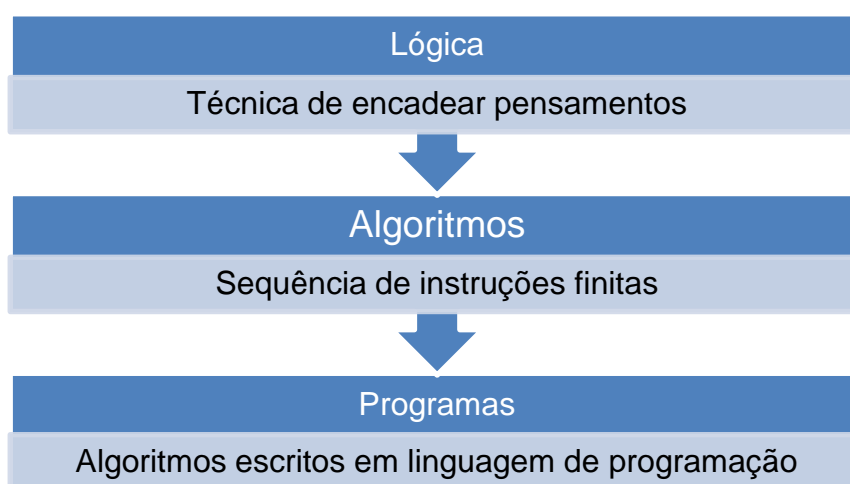
Para se desenvolver programas lógicos é necessário primeiro pensar em seu algoritmo. Um **algoritmo** é uma sequência de instruções finitas organizadas de forma lógica que tem por finalidade resolver um problema ou descrever uma tarefa. Essas tarefas não podem ser redundantes nem subjetivas na sua definição, devem ser sempre claras e precisas.

Um algoritmo espera ter dados de entrada que a princípio são dados brutos e que após a execução das instruções, gera dados de saída que são dados com alguma informação atribuída. Em um algoritmo que tem o objetivo de obter o cálculo do IMC – Índice de Massa Corpórea, as entradas seriam os dados relativos a altura e peso de uma pessoa, as operações aritméticas fazem parte do processamento do algoritmo e por fim, a saída do programa seria a informação relacionada ao valor do IMC e possíveis comparações com os índices da Organização Mundial de Saúde.

Quando se fala na palavra algoritmos não necessariamente se tem envolvidos aspectos computacionais, pois qualquer tarefa do dia a dia tem um algoritmo envolvido.

Já quando se alia a lógica de programação com um algoritmo é possível desenvolver um programa. **Programas** de computadores nada mais são do que algoritmos escritos em uma linguagem de programação (C, C#, C++, Pascal, Cobol, Java entre outras) e que são interpretados e executados em uma máquina, no caso um computador.

Figura 1: Conceitos Fundamentais para Aprendizado da Programação



Os computadores trabalham internamente com código binário, porém para facilitar a comunicação com os computadores foram criadas linguagens de programação de alto nível, dessa maneira os programadores não precisam escrever em binário as ações que são envolvidas em seu algoritmo.

A **linguagem C** é uma linguagem de programação utilizada para diversos propósitos, é portátil, modular e simples, desde que seja compreendida sua sintaxe e sejam feitos exercícios para fixar os conceitos relacionados a aplicação de cada comando.

Os programas são normalmente implementados em um ambiente integrado de desenvolvimento (IDE – *Integrated Development Environment*). Existem ambientes que são de código livre, ou seja, gratuitos como o DEV C++ e o Code Blocks.

Para se criar um primeiro programa que o computador seja capaz de executar, deve-se ter instalado um dos ambientes de desenvolvimento citados anteriormente e na sequencia deve-se criar um novo arquivo fonte. O código fonte do programa é o arquivo que será salvo contendo todas as instruções e comandos do programa.

A extensão dos códigos fontes em linguagem C é “.c”. Portanto, após digitar as instruções deve-se salvar o programa (exemplo: teste.c) e na sequencia acionar o compilador do ambiente.

O código escrito em linguagem de alto nível é convertido em linguagem de máquina pelo compilador ou interpretador. O compilador lê a primeira instrução do programa, faz uma verificação de sua sintaxe e, se não houver erro, converte-a para linguagem de máquina; segue para a próxima instrução, repetindo o processo até que a última instrução seja atingida ou até que se encontre algum erro.

Após a compilação, na mesma pasta em que foi salvo o arquivo “.c” é possível observar o arquivo executável “.exe”. Ao acionar o executável do programa pode-se iniciar a fase de testes, ou seja, para validar o algoritmo é necessário realizar testes significativos para saber se saída esperada foi obtida.

2. ESTRUTURA INICIAL DE UM PROGRAMA

A linguagem C é uma linguagem estruturada, ou seja, ela é formada por blocos que são chamados de funções.

Todo o programa deve ter uma única função principal que chama-se **main()**.

A função **main()** tem a seguinte estrutura base:

```
int main()
{

    return 0;

}
```

Um programa que não utiliza outras funções criadas pelo programador, deverá ter todo o seu algoritmo desenvolvido dentro da função **main()**. O início da **main()** é marcado por seu cabeçalho (**int main()**) e o conteúdo deve estar entre a abertura e o fechamento das chaves, isto é, as chaves delimitam o corpo da função. Toda linha do programa é uma instrução que é observada pelo compilador, logo na estrutura base da **main()** observa-se a instrução **return 0** que indica para o compilador o fim da execução do programa.

Além da **main()** outras funções podem ser utilizadas no processo de desenvolvimento dos programas, isto é, dentro da **main()**. As principais funções em C podem ser adicionadas a **main()** desde que sejam indicadas as bibliotecas onde se encontram essas funções.

Para incluir nos programas as bibliotecas com suas funções pré-definidas, deve-se utilizar o comando **#include**. O comando é seguido do nome da biblioteca entre sinais de maior (<) e menor (>).

Exemplos:

#include <stdio.h>	Funções de entrada e saída
#include <stdlib.h>	Funções tipo padrão
#include <math.h>	Funções matemáticas
#include <conio.h>	Funções manipular caracteres na tela
#include <string.h>	Funções de texto

Para se incluir duas novas funções na função principal deve-se saber em qual das bibliotecas as funções se encontram e antes do cabeçalho da main() inserir os comandos **#include**. Abaixo as funções printf() e getch() pertencem correspondentemente as bibliotecas <stdio.h> e <conio.h>.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    printf("Primeiro programa");
    getch();
    return 0;
}
```

Os programadores devem ficar atentos a alguns detalhes em relação a programação, tais como:

- Finalizar as linhas com ponto-e-vírgula (;).
- Observar a forma de como as instruções são escritas, pois a linguagem C é *case sensitive*, ou seja, letras maiúsculas são diferenciadas de letras minúsculas.
- Toda vez que for observado um novo nome dentro do programa e abertura e fechamento de chaves na sequência, trata-se de uma função. Exemplo de funções: main(), printf() e getch(). Uma função é

identificada independente de possuir parâmetros de entrada (informações dentro dos parênteses) ou não.

- Sempre salve o programa antes de compilar e sempre compile o programa antes de executar. Ao acionar a opção “compilar e executar” dos ambientes de desenvolvimento, normalmente o programa já é salvo, antes de sua execução.
- Quando ocorrer um erro de compilação, acionar um duplo clique sobre a mensagem de erro faz com que seja destacado o comando errado no programa. Se na linha indicada pelo compilador o erro não seja identificado, é necessário verificar também a linha anterior, que pode ser a responsável pelo erro, especialmente se faltar o ponto-e-vírgula (;)
- Espaços em branco dentro do código fonte podem existir. Esses espaços em branco, tabulações e pular linhas no programa correspondem a caracteres ignorados pelo compilador. Para os programadores, os espaços ajudam na organização do código, o que é chamado de endentação dentro da programação.

É possível inserir comentários dentro do programa, os quais são de auxílio para os programadores e que também são descartados pelo compilador. Os comentários de uma só linha são marcados pelo “//”, já os de múltiplas linhas são iniciados pelo “/*” e finalizados pelo “*/”.

Exemplo:

```
/*  
Programa inicial  
Aluno:  
Data:  
*/  
#include <stdio.h>  
#include <conio.h>  
int main()  
{
```



```
printf("Primeiro programa");//biblioteca stdio.h  
getch(); //biblioteca conio.h - pausa o programa  
return 0; //fecha o programa  
}
```

3. VARIÁVEIS E CONSTANTES

Os programas, algoritmos escritos em uma linguagem de programação, possuem dados que precisam ser guardados durante a execução do programa, ou seja, precisam ser “memorizados” pelo computador.

3.1. Variáveis

Variável é um espaço de memória que o programa reserva para armazenar os dados de um programa. Toda variável deve possuir um tipo e também um identificador, um nome. Normalmente, o nome da variável deve-se estar relacionado com o dado que ele irá armazenar, justamente para facilitar a programação.

Os tipos possíveis de dados nativos na linguagem C são:

Tipo do Dado	Dado	Tamanho Aproximado
int	Valores inteiros	2 bytes
char	Caracteres	1 byte
float	Valores reais	4 bytes
Double	Valores reais com precisão dupla	8 bytes
Void	-	-

Nos programas pode-se ter uma ou mais variáveis do mesmo tipo e tipos variados podem ser utilizados dependendo de qual é o escopo do programa. O tamanho aproximado é medido em bytes, que é a maneira de se especificar a quantidade de espaço de armazenamento de um dispositivo. Cada byte representa um caractere no computador e equivale a 8 bits, que é a menor unidade de informação.

Os nomes das variáveis podem ser apenas uma letra ou até uma palavra com 32 caracteres, porém não é permitido espaço caso seja um nome composto.

Exemplo:

```
preço_unitario;  
preço_total;  
valor_final;
```

O nome da variável pode conter letras e números, desde que o nome da variável não se inicie pelo número. O nome não pode também ter acento ou cedilha, por exemplo, já que são características da língua portuguesa e o compilador não entende.

Ainda sobre os nomes, não se podem usar os nomes reservados na linguagem como nome de variáveis, como os termos abaixo:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Após definir o tipo do dado que se precisa armazenar e o nome compatível com a informação, deve-se iniciar a programação dentro da função main() pela declaração das variáveis. A sintaxe para a declaração de uma variável é:

```
tipo_var nome_var;
```

Sendo assim escreve-se na sequência: tipo da variável, seu nome e ponto e vírgula.

Exemplos:

```
int num1; //variável que irá guardar um valor inteiro  
float n1, n2; //variáveis que irão armazenar valores  
reais
```

```
char resp; //variável que irá guardar um caractere
```

As variáveis, portanto tem a característica de serem mutáveis cada vez que um programa executa, porém caso sejam identificados valores que sempre serão iguais independente da quantidade de vezes que o programa execute, pode-se então utilizar as constantes.

3.2. Constantes

Constantes representam localizações de memória onde são armazenados dados que não podem ser modificados durante a execução do programa. As constantes são definidas após a declaração das bibliotecas. A sintaxe para a definição de uma constante é:

```
#define nome_const valor_const
```

Sendo assim primeiro é inserida a diretiva `#define` e depois: o nome da constante e o valor da constante sem ponto e vírgula no final da linha.

Exemplos:

```
#define PI 3.1415 //constante matemática
#define MAX 100 //constante para um valor máximo
#define OP 'S' //constante para definir um caractere
```

Abaixo segue um exemplo onde foram apenas definidas variáveis e constantes para o cálculo da área de um círculo.

```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1415 //constante PI
int main() //a main() é tipo inteiro - int
{
    //grandezas físicas são normamente tipo float
    //raio é uma variável de entrada desse programa
    //área é uma variável de saída desse programa
```

```
float raio, area;  
printf("Area de um circulo");  
getch();  
return 0;  
}
```

4. COMANDOS DE ENTRADA E SAÍDA

Para mostrar textos ou mesmo valores das variáveis na tela do computador utiliza-se a função **printf()**, já para receber os valores digitados no teclado e enviá-los para um dos espaços reservados para uma variável utilizamos o comando **scanf()**.

As funções `printf()` e `scanf()` são portanto funções de entrada e saída (in/out) e estão descritas na biblioteca `<stdio.h>`.

4.1. Função `printf()`

O **printf()** caracteriza-se por ser um comando de saída formatada, ou seja, pode-se ter um ou mais argumentos dentro dos parênteses da função.

A sintaxe do `printf()` é: `printf(a1,a2,a3,...,an);`

O primeiro argumento (`a1`) chama-se string de formato, deve estar entre aspas duplas e pode conter:

- Texto: `printf("Boa noite!");` //Frase Boa noite! será exibida
- Códigos de barra invertida: `printf("Boa \n noite!");`
- Especificadores de formato: `printf("O valor de x é %i",x);`

Os demais argumentos (`a2,a3, ..., an`) chamam-se itens de dados e são quantos forem os especificadores de formato do primeiro argumento

Exemplos:

```
printf("Peso= %f - idade= %i - sexo= %c", kg, id, sexo);  
printf("Resultado = %.2f ", num);
```

Os códigos de barra invertida podem tornar a saída dos programas mais agradável, ou seja, formatando os dados da maneira que se julgar necessário.

Segue abaixo os principais códigos de barra invertida:

Códigos	Formato
\n	Nova linha
\t	Tabulação (tab)
\a	Beep – Toque do auto-falante
\\	Barra invertida
\0	Zero
\'	Aspas simples (apóstrofo)
\"	Aspas duplas.
\xdd	Representação hexadecimal
\ddd	Representação octal

Os códigos de formatação, também chamados de especificadores de formato são utilizados para determinar o tipo de dado que será lido ou escrito na tela. Os códigos de formatação são mostrados abaixo:

Códigos	Tipo de Dados
%c	char (caracteres individuais)
%s	char (conjunto de caracteres)
%d ou %i	int
%f	float
%lf	double
%e	Número em notação científica, por exemplo, 10.3e-8
%x	Número em hexadecimal, por exemplo, 32F2
%p	Endereço de memória (ponteiro)
%%	Imprime um %

4.2. Função scanf()

A função **scanf()** é utilizada para ler dados do teclado e colocar os valores fornecidos pelo usuário nas variáveis utilizadas como parâmetros da função.

A sintaxe do scanf() é: `scanf(a1, a2, a3, ..., an);`

O primeiro argumento (a1) chama-se string de formato, deve estar entre aspas duplas e pode conter:

- Especificadores de formato: `scanf("%i",&x);` //recebe um valor inteiro - %i

Os demais argumentos (a2,a3, ... ,an) chamam-se itens de dados. São quantos forem os especificadores de formato do primeiro argumento e devem ser precedidos pelo operador **& (endereço de)**. Caso o `scanf()` não possua o operador de endereço, o programa pode finalizar inesperadamente, já que o compilador não consegue saber onde irá guardar o valor recebido através do teclado.

Exemplos:

```
scanf("%c", &resp); //receber valor char no endereço da
variável resp
scanf("%f %i %c", &kg, &id, &sexo);
```

No exemplo acima, quando o programa executa a função `scanf()` ele espera receber três valores de entrada separados por espaços. O primeiro valor é tipo real (float) e seu valor será armazenado no endereço da variável *kg*, o valor int no endereço da variável *id* e o valor char no endereço da variável *sexo*.

Inserindo as funções de entrada e saída no programa do cálculo da área de um círculo, o programa ficará da seguinte forma:

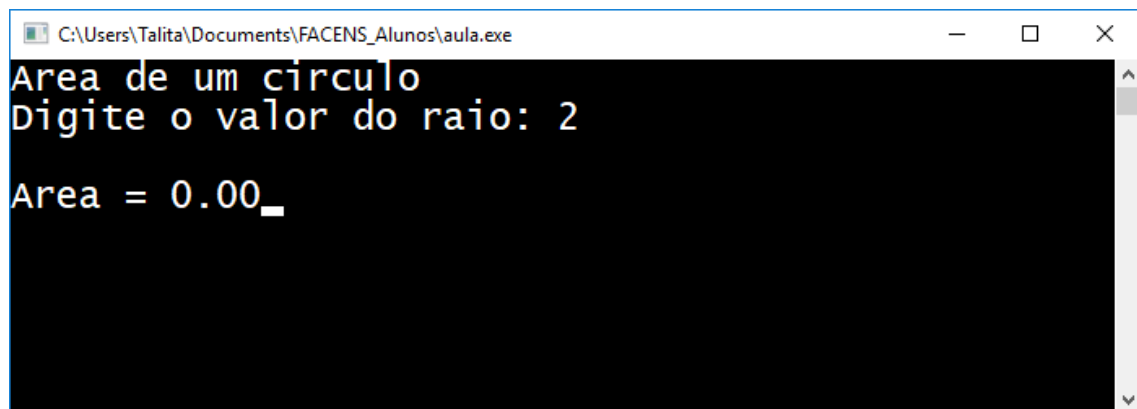
```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1415 //constante PI
int main()
{
    float raio, area;
    printf("Area de um circulo\n"); //Pula linha
    printf("Digite o valor do raio: ");
    scanf("%f",&raio);
    //Calcular a área do circulo para mostrar o
    resultado
```



```
//Exibir apenas 2 casas decimais no resultado - %.2f
//Usar %f mostra 6 casas decimais como padrão
printf("\nArea = %.2f",area);//valor da variável
area

    getch();
    return 0;
}
```

Na tela é mostrado primeiro o texto dos dois printf(), na sequência o cursor fica piscando esperando que um valor numérico seja digitado, após a digitação do valor 2, como mostrado abaixo, o valor é guardado na variável raio, e é mostrado o texto da saída final, porém como ainda não foi realizado o cálculo da área, o valor mostrado como conteúdo da variável *area* é zero.



5. OPERAÇÕES

A linguagem C é muito rica em operadores internos e as principais classes de operadores são: aritméticos, relacionais e lógicos.

5.1. Operador de Atribuição

O **operador de atribuição**, representado pelo símbolo “=”, é utilizado quando se deseja atribuir um valor a uma variável. Quando se declara uma variável, por exemplo, pode-se também já atribuir um valor a ela. Em relação a atribuição deve-se apenas verificar se a variável e o valor atribuído sejam de tipos compatíveis para que não ocorram erros.

Exemplos:

```
int a=10;//declaração e inicialização da variável  
num=3.5 //atribuição do valor 3.5 a variável num  
opcao='S'//atribuição do caractere S a variável opcao
```

Observação: as casas decimais de valores float são marcadas por ponto e não vírgula, seja no momento de digitar um valor de entrada na execução do programa ou para inserir cálculos no código fonte do programa.

5.2. Operadores Aritméticos

Os **operadores binários** são aqueles usados em operações que possuem dois operandos. Na linguagem C, os operadores binários são:

Operador	Operação
+	Adição
-	Subtração
* - asterisco	Multiplicação
/	Divisão
% - percentual	Módulo (resto da divisão inteira)

Os operadores de adição, subtração e divisão não sofrem alterações dentro do que já se conhece das operações básicas elementares. O asterisco sinaliza a multiplicação dentro dos cálculos. O % representa o resto da divisão dos valores inteiros, ou seja, esse operador somente pode ser usado entre operandos de tipo inteiro.

Exemplo:

```
int a=10, b=3, resto;  
resto=a%b;
```

No exemplo acima, o valor inteiro 10 dividido por 3 é 3, porém existe um resto que é igual a 1. O valor 1, portanto é atribuído a variável resto.

Importante: caso seja necessário calcular o valor percentual de um valor, deve-se utilizar a operação de multiplicação.

Exemplo: desconto de 10% do valor total de um compra.

```
float valor, valor_desconto;  
valor_desconto=valor*0.9;
```

ou

```
valor_desconto=valor-valor*0.1;
```

Outra questão importante no uso de operadores é quando eles são usados em conjunto, ou seja, em uma só expressão. Nesse caso utiliza-se a seguinte relação de precedência:

Ordem	Operador
1º	Parênteses mais internos
2º	Potenciação e raiz
3º	* e /
4º	+ e -

Exemplo: média aritmética entre dois valores reais

```
float media, n1=1.5, n2=2.0;  
media=(n1+n2)/2;
```

A linguagem C também oferece funções matemáticas predefinidas:

Para valores inteiros (`stdlib.h`):

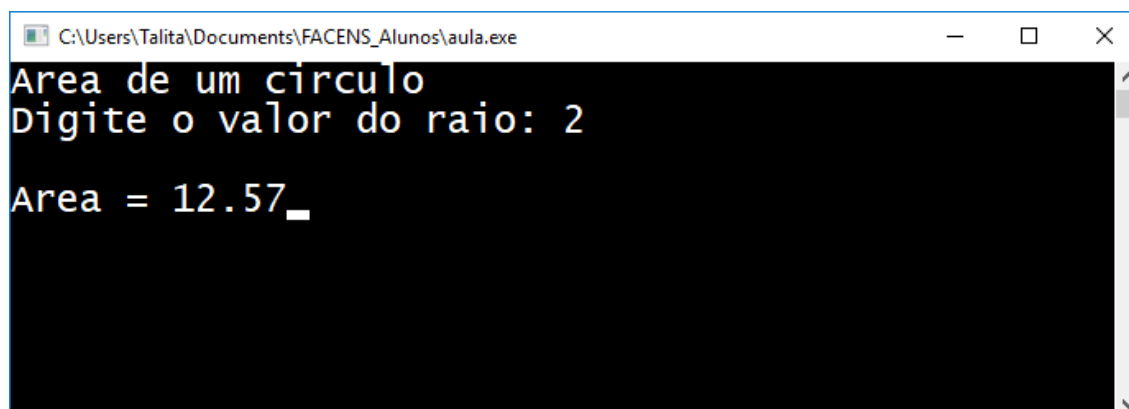
- `abs(x)` : retorna o valor absoluto de x.

Para valores `double` (`math.h`):

- `atan(x)` : retorna arco tangente de x.
- `cos(x)` : retorna cosseno de x.
- `sin(x)` : retorna seno de x.
- `exp(x)` : retorna função exponencial natural (e^x).
- `pow(x, y)` : eleva x à potência y.
- `sqrt(x)` : retorna a raiz quadrada de x.
- `log(x)` : retorna o logaritmo natural de x.

Conhecendo os operadores aritméticos, funções matemáticas e o operador de atribuição pode-se agora finalizar o programa do cálculo da área de um círculo.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h> //para usar função pow()
#define PI 3.1415
int main()
{
    float raio, area;
    printf("Area de um circulo\n"); //Pula linha
    printf("Digite o valor do raio: ");
    scanf("%f", &raio);
    //Processamento
    area = PI * pow(raio, 2);
    printf("\nArea = %.2f", area);
    getch();
    return 0;
}
```



A linguagem C além dos operadores binários possuem também os operadores unários. Os **operadores unários** atuam em apenas um operando e são eles: menos unário, incremento e decremento.

Operação	Operador	Exemplo
Menos Unário	-	$-x \rightarrow x * (-1)$ //Multiplica a variável por -1
Incremento	++	$x++ \rightarrow x = x + 1$ //Incrementa a variável depois de usá-la
Decremento	--	$x-- \rightarrow x = x - 1$ //Decrementa a variável depois de usá-la

Pode-se utilizar o operador de incremento e decremento antes da variável, nesse caso, a variável será incrementada/decrementada antes de usar a variável. Utiliza-se com frequência esses operadores na estrutura de repetição FOR.

5.3. Operadores Relacionais

Os operadores relacionais são utilizados para comparar variáveis ou expressões e resultam em verdadeiro ou falso.

Na linguagem C os operadores relacionais são:

Operador	Símbolo
Igualdade	==
Diferença	!=
Maior que	>
Menor que	<
Maior ou igual que	>=
Menor ou igual que	<=

5.4. Operadores Lógicos

Os operadores lógicos são usados para relacionar duas expressões ou apenas uma que é o caso do operador de negação.

- **Operador AND** – representador pelo símbolo `&&` (e comercial duplo)
Resulta em verdadeiro se ambas expressões relacionais forem verdadeiras
- **Operador OR** – representador pelo símbolo `||` (barra vertical dupla)
Resulta em verdadeiro se pelo menos uma expressão relacional for verdadeira
- **Operador NOT** – representador pelo símbolo `!` (exclamação)
Resulta em verdadeiro se a expressão relacional for falsa e falso, se a expressão for verdadeira.

No capítulo a seguir, sobre estrutura de decisão, será possível visualizar exemplos tanto de operadores lógico quanto operadores relacionais nas expressões utilizadas.

6. ESTRUTURA DE DECISÃO – IF

Uma das tarefas fundamentais de qualquer programa é decidir o que deve ser executado a seguir. Os **comandos de decisão** ou comandos de seleção permitem que o programador possa alterar a sequência de execução de um programa.

Dessa forma, surge a possibilidade de selecionar entre ações alternativas, dependendo dos critérios desenvolvidos, o que será executado no decorrer do programa.

6.1. Comando if

O comando **if** (se) representa uma tomada de decisão simples. A sintaxe do comando if é:

```
if (condicao)//não tem ; porque o comando if termina em {  
    {  
        //instruções  
    }
```

Quando utiliza-se uma instrução if, apenas expressões com resultados booleanos (verdadeiro ou falso) podem ser avaliadas. A condição usando um operador relacional deve estar entre parênteses e após o comando if. Se a condição do if for verdadeira, as instruções do comando if (entre as chaves) serão executadas, já se a condição for falsa, nenhuma instrução das chaves será executada e o programa segue executando as instruções subsequentes.

O uso das chaves do comando if são opcionais se existe apenas uma instrução a ser executada, caso o comando if tenha mais instruções é essencial o uso das chaves para que o compilador compreenda quais as instruções que pertencem aquele comando.

Importante: Para usar o if não é necessário adicionar uma nova biblioteca.

Exemplos:

```
int a=10, b=5;
if(a==10)//expressão verdadeira,executa os comandos do if
if(a!=b) //expressão verdadeira,executa os comandos do if
if(a<=b) //expressão falsa, não executa os comandos do if
if(a>b && !(a>20)) //expressão verdadeira
if(a<b || a<20) //expressão verdadeira
{
    printf("Valor a \x82 menor que b ou menor que 20");
    //\x82 representação hexadecimal do é (e com acento)
}
```

6.2. Comando if else

O comando **else** pode ser usado opcionalmente após o if. Se a condição do if é verdadeira, é executado o conteúdo de dentro do if, caso contrário, é executado o conteúdo de dentro do else. A sintaxe nesse caso é:

```
if (condicao) //sem ponto e vírgula após a expressão
{
    //instruções
}
else //sem ponto e vírgula e sem condição de teste
{
    //instruções
}
```

O comando **if else** (se/senão) garante que uma das duas opções sejam executadas, nunca as duas ou nenhuma delas.

É importante lembrar que o comando `else` não possui uma condição de teste, já que ele representa uma condição contrária a condição do `if`. Caso seja colocado um teste na frente do comando `else` ocorrerá um erro de compilação.

6.3. Comando `if else` aninhados

Quando existe a situação de ter mais de dois caminhos a seguir durante a execução do programa, uma estrutura de decisão `if else` simples não resolve o problema. Nestes casos pode-se utilizar estruturas de `if else` aninhadas.

A **estrutura de decisão aninhada** utiliza a mesma sintaxe de uma estrutura de decisão simples. A denominação “aninhada” refere-se a ter uma estrutura dentro da outra.

O objetivo de aninhar estruturas de decisão é permitir a implementação de problemas mais complexos, logo, a indentação do código fonte auxilia na compreensão do programa e deve ser sempre utilizada, mesmo sabendo que o compilador não a leva em consideração.

Exemplo:

```
float num=4.5;
if (num > 10)
{
    printf("O numero %.2f \x82 maior que 10.", num);
}
else
{
    if(num <10)
    {
        printf("O numero %.2f \x82 menor que 10.", num);
    }
}
```

```
        else
        {
            printf("O numero digitado \x82 igual a 10.");
        }
    } //fim do else externo
```

Nesse caso o programa executará a primeira condição, if externo, e ela será falsa, logo a execução segue para o else e sua primeira instrução é um novo if. Esse novo if retornará uma condição verdadeira, portanto será mostrado na tela o texto: O numero 4.5 é menor que 10. O programa então segue para as instruções após o comando if else aninhado, pois a última instruções (else interno) não será executada.

O mesmo trecho do programa acima pode ser escrito usando logo na frente do else o “if interno”, ou seja, nesse caso costuma-se chamar de sintaxe “else if”. Segue o mesmo exemplo a seguir com essa organização, lembrando que para o compilador não há diferenciação entre as formas apresentadas.

Exemplo:

```
float num=4.5;
if (num > 10)
{
    printf("O numero %.2f \x82 maior que 10.",num);
}
else if(num <10)
{
    printf("O numero %.2f \x82 menor que 10.",num);
}
else
{
    printf("O numero digitado \x82 igual a 10.");
} //fim do else interno
```

7. ESTRUTURA DE SELEÇÃO – SWITCH

O **switch** é um comando de seleção múltipla e tem como objetivo testar sucessivamente o valor de uma expressão contra uma lista de constantes inteiras ou de caracteres.

O comando switch só pode testar igualdade (==), enquanto que o if pode avaliar uma expressão lógica e/ou relacional (< , > , <= , >= , != , ! , && , ||).

A sintaxe do comando switch é:

```
switch(variável)
{
    case constante1:
        //instruções
        break;
    case constante2:
        //instruções
        break;
    default:
        //instruções
}
```

O comando **case** representa cada teste de igualdade que será realizado em relação a variável entre parentes após o comando switch. Caso a variável tenha o valor de um dos cases, as instruções do determinado case serão executadas. Na sequência o comando break será executado.

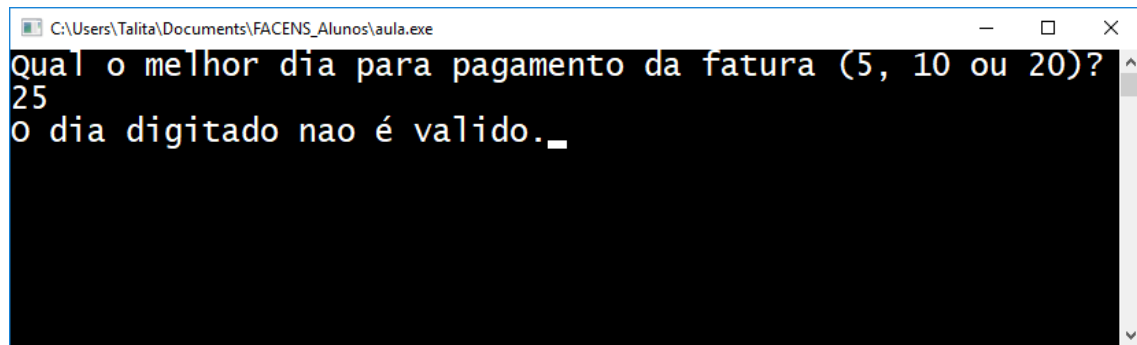
O comando **break** é um dos comandos de desvio em C. Quando um break é encontrado em um switch, a execução do programa “salta” para a linha de código seguinte ao comando switch.

O comando **default** é opcional e as instruções pertencentes a ele serão executadas, caso nenhuma das constantes testadas tenha o valor da variável do switch.

Exemplo:

```
int dia;
printf("Qual o melhor dia para pagamento da fatura (5, 10
ou 20)?\n");
scanf("%i",&dia);
switch(dia)
{
    case 5:
        printf("O dia 5 foi escolhido com sucesso.");
        break;
    case 10:
        printf("O dia 10 foi escolhido com sucesso.");
        break;
    case 20:
        printf("O dia 20 foi escolhido com sucesso.");
        break;
    default:
        printf("O dia digitado nao \x82 valido.");
}
```

No caso acima, se for digitado o dia 10, o programa mostrará a mensagem afirmativa que a escolha foi realizada com sucesso. Se quaisquer valores diferente de 5, 15 e 20 forem digitado será mostrada a mensagem de erro, ou seja, a mensagem “default” (padrão).



```
C:\Users\Talita\Documents\FACENS_Alunos\aula.exe
qual o melhor dia para pagamento da fatura (5, 10 ou 20)?
25
O dia digitado nao é valido._
```

8. ESTRUTURA DE REPETIÇÃO – FOR

As estruturas de repetição são também chamadas de laços. Os **laços** são comandos usados sempre que uma ou mais instruções tiverem de ser repetidas enquanto uma certa condição estiver sendo satisfeita.

Em C existem três comandos de laços: **for**, **while** e **do-while**.

O comando **for** é utilizado quando já se conhece a quantidade de repetições que irão ocorrer. Exemplo: calcular a média dos trabalhos de uma turma de 40 alunos, ou seja, nesse caso sabe-se previamente que o procedimento do cálculo da média deverá ser executado 40 vezes.

A sintaxe do comando **for** é:

```
for (inicialização; condicao; incremento)
{
    //instruções
}
```

O laço **do for** utiliza uma variável inteira para controlar a quantidade de repetições. Essa variável utilizada no laço é chamada de variável de controle.

No primeiro termo de parametrização do **for** é realizada a inicialização da variável de controle. A inicialização ocorre através do operador de atribuição (=) e ela é executada apenas uma única vez, antes que as repetições iniciem.

A condição (teste) é avaliada como verdadeira ou falsa, portanto deve conter um ou mais operadores lógicos ou relacionais. Se a condição for verdadeira, as instruções entre chaves serão executadas. Já se a condição for falsa, o laço é encerrado e o programa segue para a próxima instrução. Se após a inicialização for verificado que a condição é falsa, a repetição das instruções não ocorrerá.

A avaliação da condição ocorre toda vez que o **for** é inicializado ou reinicializado.

O último termo do **for** é o incremento ou decremento (operadores unários). Eles definem a maneira pela qual a variável de controle será alterada cada vez que o laço **for** for repetido. Essa expressão é sempre executada logo após a execução do corpo do laço.

Ainda sobre a sintaxe, os termos de parametrização do **for** devem estar entre parênteses e são separadas por ponto-e-vírgula. Após o fechamento dos parênteses não há pontuação, inicia-se o bloco de instruções do **for**, marcado pela abertura e fechamento das chaves.

Exemplos:

```
//Mostrar 10 vezes a letra A, sendo uma vez por linha
int x; //variável de controle do for
for(x=1;x<11;x++) //10 repetições de 1 printf()
{
    printf("A\n");
}

//Receber 3 números e dizer se cada um é par ou não
int i, num; //i é a variável controle do for
for(i=0;i<3;i++) // 3 repetições
{
    printf("\nDigite o %i\xF8 numero: ",i+1); //\xF8 é °
    scanf("%i",&num);
    if (num%2==0)//se ao dividir por 2 o resto é 0 é par
    {
        printf("O numero %i \x82 par.",num);
    }
    else
    {
        printf("O numero %i nao \x82 par.",num);
    }
}
} //fim do for
```

```
//Calcular a média dos trabalhos de 40 alunos
int aluno; //variável controle do for
float t1, t2, media;
for(aluno=1;aluno<=40;aluno++)
{
    printf("\nDigite nota 1 e nota2 do aluno
%i:",aluno);
    scanf("%f %f",&t1,&t2);//receber duas variáveis
    media=(t1+t2)/2;
    printf("Aluno %i - Media= %.2f",aluno,media);
}
```

Assim como o comando if else, os laços de repetição também podem ser usados de forma **aninhada**, desde que seja declarada uma variável inteira de controle para cada laço for existente no programa.

Nos dois últimos exemplos mostrados acima, observa-se que ao se usar apenas uma variável para receber os valores ao longo das repetições, a cada nova repetição faz com que o valor atual da variável seja sobrescrito. Isso ocorre, pois ainda não é possível guardar conjuntos de dados do mesmo tipo de uma maneira unificada.

Nesses casos, quando se deseja obter informações sobre o conjunto de dados, deve-se fazer verificações e memorizar informações ao longo das repetições. Para isso, pode-se utilizar outras **variáveis auxiliares** no programa, algumas dessas auxiliares tem uma denominação já estabelecida para facilitar a programação, são as variáveis **acumuladoras** e **contadoras**.

As variáveis auxiliares são declaradas da mesma forma que as demais variáveis do programa, porém o tipo da variável deve estar de acordo com a sua função no programa. Uma variável acumuladora ou totalizadora, por exemplo, deve ser do mesmo tipo do dado que se deseja acumular. Já as variáveis contadoras devem ser necessariamente do tipo inteiro, pois se trata

da contagem de ocorrências de um determinado evento durante a execução do programa. As variáveis auxiliares normalmente são inicializadas em suas declarações, ou seja, permitindo que as variáveis estejam prontas para o uso com valores já definidos.

Exemplos:

```
int total=0;//Variável acumuladora total inicializada em 0
int cont=0;//Variável contadora cont inicializada em 0
float aux=10;//Variável auxiliar inicializada em 10
```

No exemplo abaixo é possível observar uma variável acumuladora chamada de “soma” que tem o objetivo de somar os valores digitados a medida que eles são digitados e inseridos na variável “num”. Dessa forma, existem somas parciais durante as repetições do for, porém somente após o fechamento do for é mostrado o resultado final da somatória. No caso, a variável soma é int, porque está acumulando valores inteiros, se fosse para valores reais, a variável soma deveria ser tipo float.

Exemplo:

```
//Mostrar a soma de 5 valores inteiros digitados
int x, num, soma=0; //soma é a variável acumuladora
for(x=0;x<5;x++)
{
    printf("Digite o %i\n",x+1);
    scanf("%i",&num);
    soma+=num;//mesmo significado de soma=soma+num;
}
printf("Soma = %i",soma);
```

A variável contadora mostrada abaixo tem o objetivo de ir contando, a medida que as repetições ocorrem, a quantidade de vezes em que se tem um valor par digitado. Assim como no exemplo anterior existem “contagens” parciais e depois se verifica a saída final, após o for, para ver se está de acordo com o esperado.

Exemplo:

```
//Mostrar "quantos" dos 5 valores digitados são pares
int x, num, cont=0; //cont é a variável contadora
for(x=0;x<5;x++)
{
    printf("Digite o %i\xF8 numero: ",x+1);
    scanf("%i",&num);
    if (num%2==0)
    {
        cont++; //soma 1 na variável auxiliar cont
    }
} //fim do for
printf("Quantidade de numeros pares = %i",cont);
```

É comum também variáveis auxiliares para se obter alguma informação específica durante a execução do programa. Como é o caso do exemplo abaixo, onde se deseja obter o menor valor digitado entre um conjunto de números digitados.

Exemplo:

```
//Mostrar qual o menor valor digitado
int x, num, menor; //menor é a variável auxiliar
for(x=0;x<5;x++)
{
    printf("Digite o %i\xF8 numero: ",x+1);
    scanf("%i",&num);
    /*se num for menor que o conteúdo da variável menor
    ou se é a primeira repetição, inicializa menor com o
    primeiro valor de num*/
    if (num<menor || x==0)
    {
        menor=num; //atualiza a auxiliar menor
    }
}
```

```
//fim do for  
printf("Menor valor digitado = %i",menor);
```

9. ESTRUTURA DE REPETIÇÃO – WHILE e DO-WHILE

Os comandos **while** e **do-while** são utilizados quando não se conhece a quantidade de repetições que irão ocorrer durante a execução do programa. Nesse caso, dependendo de ações ou condições desenvolvidas ao longo da repetição, pode-se definir se ela irá ocorrer ou não. Exemplo: mostrar quantos números são múltiplos de 3 a partir de um intervalo definido na execução do programa. Nesse caso, como não se conhece o intervalo a ser digitado como entrada do programa, não se torna possível saber quantos serão os números apresentados como múltiplos de 3.

9.1. Repetição While

O primeiro comando de repetição que não exige a quantidade de repetições antes do seu início é o **while** (enquanto). A sintaxe do comando **while** é:

```
while (condicao)
{
    //instruções
}
```

Assim como no laço **for**, enquanto a condição em parênteses estiver verdadeira, as instruções que estão dentro das chaves serão executadas, caso contrário não serão executadas as repetição. Se a condição, já for falsa na primeira vez que o compilador irá testá-la, nenhuma vez será executado o conteúdo das chaves.

Exemplos:

```
//Números múltiplos de 3 em um determinado intervalo
int inicio, fim, n;
printf("Digite inicio e fim do intervalo: ");
scanf("%i %i", &inicio, &fim);
```

```
n=inicio;//inicializando n
while (n<fim)//enquanto n menor que o fim do intervalo
{
    if(n%3==0)//se o n for múltiplo de 3
    {
        printf("\n%i",n);//mostrar um valor por linha
    }//fim do if
    n++;//incrementar para passar para o próximo número
}//fim do while
```

```
//Permitir digitar números enquanto o zero não for
digitado
float num=-1;
while (num!=0)//enquanto numero for diferente de zero
{
    printf("Digite um numero: ");
    scanf("%f",&num);
    if (num==0)
    {
        printf("Processo finalizado");
    }
}
}//fim do while
```

```
/*Foi realizada uma aplicação R$ 1000,00 com rendimento de 5% ao mês. Quantos meses serão necessários para o capital investido ultrapasse a R$ 2000,00?*/
```

```
float valor = 1000.00;
int cont=0;//contagem dos meses
while(valor<=2000.00)
{
    valor+=valor*0.05;//somatório de mais 5% ao mês
    cont++;
}
printf("Serao      %i      meses      para      obter      %.2f
reais",cont,valor);
```

A estrutura de repetição **while** também pode ser usada de forma aninhada e em conjunto com todos os outros comandos já estudados. A estrutura de repetição **for** pode ser substituída por um **while**, ou seja, utilizando os mesmos elementos (inicialização, condição e incremento), mas esses são distribuídos de maneira diferente no programa.

Exemplo:

```
//Mostrar 10 vezes a palavra FACENS, sendo uma vez por linha
int x=1;//variável de controle do while inicializada em 1
while(x<11) //enquanto o teste for verdadeiro mostra 'B'
{
    printf("FACENS\n");
    x++; //incrementa variável de controle
}
```

9.2. Repetição Do-While

O comando **do-while** é uma variação do while, cuja a diferença é que a condição é executada após o laço, ou seja, **ao menos uma vez** as instruções são executadas e depois é verificada a condição de teste. Se a condição de teste for verdadeira, uma nova repetição deve ocorrer a partir do comando “do”, caso contrário, não serão realizadas mais repetições.

O do-while é usado, portanto, em situações em que se tenha um menu de opções ou mesmo quando ainda não se conhece o valor da variável que será testada, sendo assim é preciso recebê-la primeiro e depois verificar (testar) o seu conteúdo.

A sintaxe do comando do-while é:

```
do
{
    //instruções
} while (condicao);
```

A partir do comando **do** (faça) segue-se o bloco de instruções entre parênteses. Após os parênteses se encontra o comando while, sua condição de teste e o ponto-e-vírgula, o qual indica a finalização da estrutura de repetição.

A estrutura de repetição **for** pode ser substituída por um **do-while**, porém o inverso não é verdadeiro, já que o for mantém verificando sua condição de teste antes de inicializar a repetição enquanto no do-while a condição é verificada sempre depois.

Exemplo:

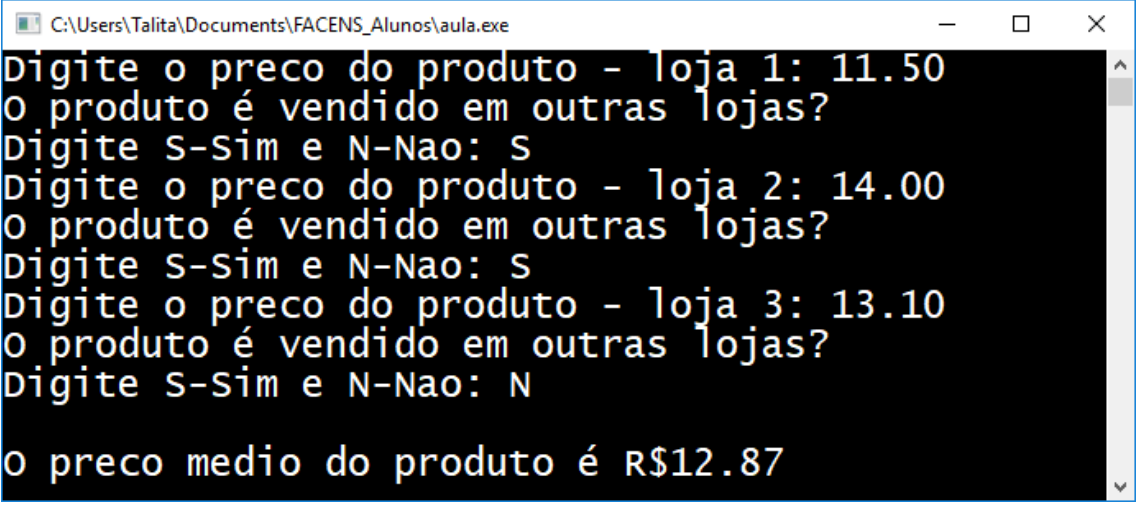
```
//Simulação de operações em uma conta bancária
float valor, saldo=0;
int op;
do
{
```

```
printf("Menu:\n[1]Depositar\n[2]Retirar\n[3]Sair");
printf("\nDigite sua opcao: ");
scanf("%i",&op);
if(op<=2)
{
    printf("Digite o valor (reais): ");
    scanf("%f",&valor);
    switch(op)
    {
    case 1:
        saldo+=valor;
        printf("Saldo atual = %.2f\n",saldo);
        break;
    case 2:
        saldo-=valor;
        printf("Saldo atual = %.2f\n",saldo);
        break;
    default:
        printf("Opcao invalida\n");
    }
}
}

//Calcular o preço médio de um produto em diversas lojas
int i=0;
float valor, soma=0;
char resp;
do
{
    i++;
    printf("Digite o preco do produto - loja %i: ",i);
    scanf("%f", &valor);
}
```



```
soma+=valor;//acumulando os valores
printf("O produto \x82 vendido em outras lojas?");
printf("\nDigite S-Sim e N-Nao: ");
scanf(" %c",&resp); //deixar espaço entre " e %c
}while(resp=='S'); //aspas simples no caractere S
printf("\nO preco medio do produto \x82 R$%.2f",soma/i);
```



```
C:\Users\Talita\Documents\FACENS_Alunos\aula.exe
Digite o preco do produto - loja 1: 11.50
O produto é vendido em outras lojas?
Digite S-Sim e N-Nao: S
Digite o preco do produto - loja 2: 14.00
O produto é vendido em outras lojas?
Digite S-Sim e N-Nao: S
Digite o preco do produto - loja 3: 13.10
O produto é vendido em outras lojas?
Digite S-Sim e N-Nao: N
O preco medio do produto é R$12.87
```

10. VETORES

No decorrer do curso já foram estudados os tipos de dados, operações, estrutura de controle e de repetição, e variáveis auxiliares. Com esses recursos são possíveis uma série de aplicações, porém para cada valor a ser memorizado durante o programa era necessário criar uma variável, as quais eram alocadas de forma dispersa na memória do computador.

Para facilitar o trabalho de processar um conjunto de dados do mesmo tipo, surge a possibilidade de usar vetores nos programas. Um **vetor** é um conjunto de elementos armazenados em uma sequência contínua de memória. **Elemento** é o nome dado para cada variável do conjunto de variáveis que compõe o vetor.

Os elementos do vetor são todos do **mesmo tipo**, o vetor tem um nome único, mas os elementos podem ser acessados individualmente a partir da sua posição ou índice.

O vetor é chamado de unidimensional quando possui apenas um índice para acessar um conteúdo. A capacidade de um vetor é fixa e deve ser informada quando se declara o vetor.

Para declarar o vetor, além do tipo e o nome, como ocorre com as variáveis simples, deve-se informar a sua capacidade. A sintaxe da declaração de um vetor unidimensional é:

```
tipo_var nome_var[quantidade_de_elementos];
```

Sendo assim, a quantidade de elementos deve estar entre os colchetes e deve ser representado por um valor inteiro. Esse valor inteiro pode ser o valor numérico mesmo ou até uma constante, desde que esta seja inteira e represente um valor numérico.

Exemplos:

```
float notas[40];
```

```
int sorteio[6];  
char palavra[20];
```

Após a declaração, pode-se, portanto acessar cada elemento do vetor para: guardar valores já definidos, guardar valores digitados (scanf()), mostrar os valores e obter informações através dos valores.

Para declarar e inicializar os valores de um vetor, pode-se usar um conjunto de inicialização, onde os valores estarão entre chaves e separados por vírgulas. Nesse caso, quanto o conjunto de inicialização tem a mesma quantidade de elementos do vetor, obtém-se uma **inicialização total**. Na **inicialização parcial**, onde o conjunto inicialização tem menor valor que quantidade de elementos, os valores não inicializados recebem o valor zero.

Exemplo:

```
float notas[40]={0.0}; //Valores do vetor inicializados em 0  
int sorteio[6]={1,4,17,29,37,58}; //Inicialização total  
char palavra[20]={'a','b','c'}; //Inicialização parcial
```

Para acessar os valores do vetor é sempre necessário colocar o nome do vetor e a posição em que se tem interesse de trabalhar. O primeiro elemento do vetor tem como índice sempre o valor **zero**. A linguagem C não verifica se o índice usado está dentro dos limites válidos. Logo, se não for tratado adequadamente os limites, corre-se o risco de sobrescrever variáveis ou até mesmo uma parte do código do programa, o que pode acarretar resultados imprevisíveis na execução do programa.

Exemplo:

```
notas[0]=9.5; //atribuindo 9.5 a primeira posição do vetor  
notas[39]=8.0; //atribuindo 8.0 a última posição do vetor  
int i; //variável controle do for para percorrer o vetor  
for(i=0; i<6; i++) //digitando valores para o vetor sorteio  
{  
    printf("Digite um numero [%i]: ", i);  
    scanf("%i", &sorteio[i]); //a variável i indica índice
```

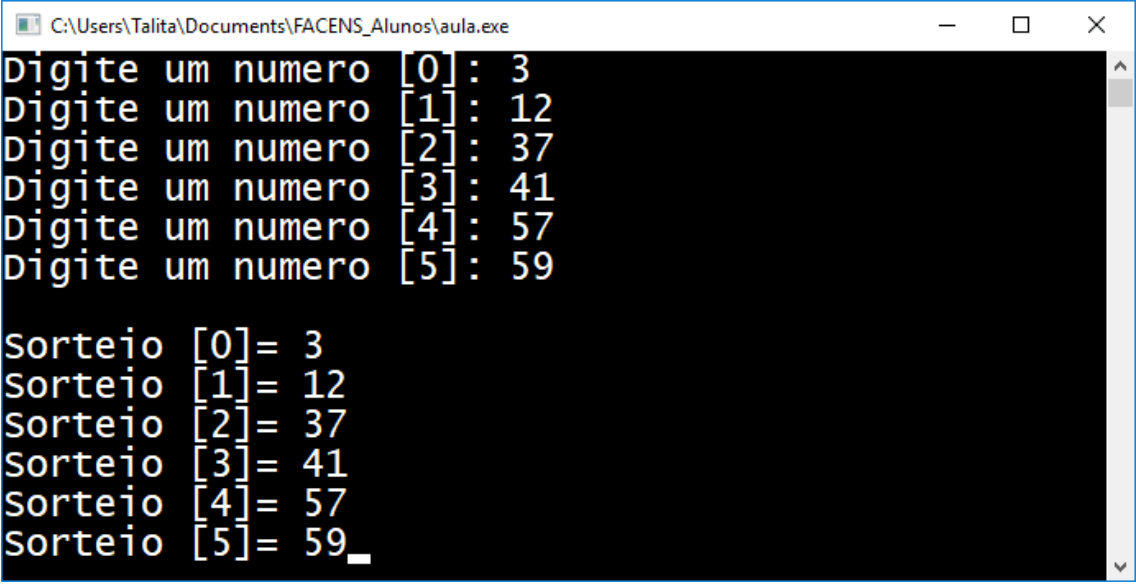
```
}
```

No exemplo, acima o vetor “sorteio” tem 6 elementos, logo seu índice inicia-se em 0 e vai até 5. Isso também é o que ocorre com a variável “i”, que é a variável de controle do **for**, logo utiliza-se a variável “i” para obter as 6 repetições e aproveita-se a mesma variável para acessar as posições do vetor.

Para mostrar o conteúdo do vetor sorteio, pode-se utilizar a mesma estrutura de repetição acima.

Exemplo:

```
//mostrando os valores contidos no vetor sorteio
for(i=0;i<6;i++)
{
    printf("\nSorteio [%i]= %i",i, sorteio[i]);
}
```



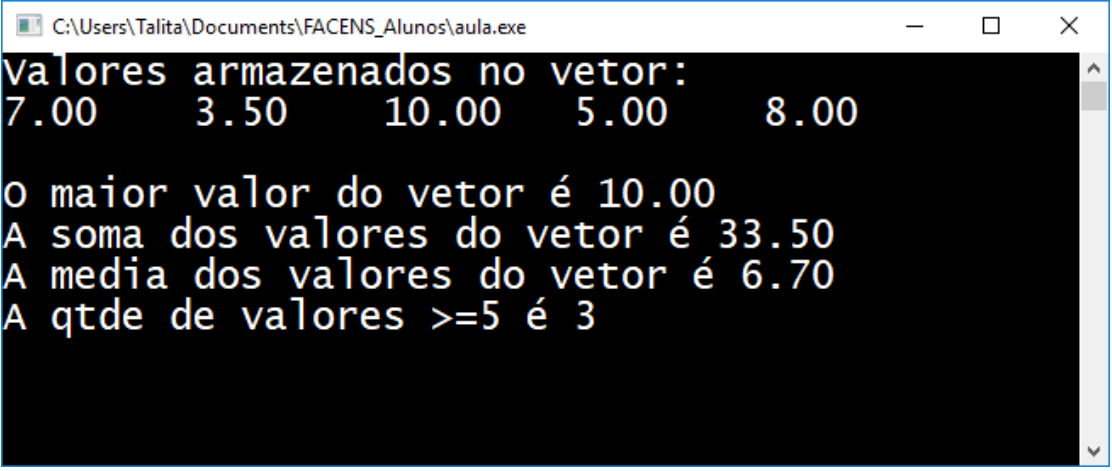
```
C:\Users\Talita\Documents\FACENS_Alunos\aula.exe
Digite um numero [0]: 3
Digite um numero [1]: 12
Digite um numero [2]: 37
Digite um numero [3]: 41
Digite um numero [4]: 57
Digite um numero [5]: 59

Sorteio [0]= 3
Sorteio [1]= 12
Sorteio [2]= 37
Sorteio [3]= 41
Sorteio [4]= 57
Sorteio [5]= 59_
```

Pode-se também obter informações relevantes dos valores contidos dentro do vetor, seja ele preenchido através do processo de inicialização ou através da digitação de valores como mostrado acima.

Exemplo:

```
int i, cont=0;//inteiros variável para índice e contadora
float vetor[5]={7, 3.5, 10, 5, 8}, maior, soma;
printf("Valores armazenados no vetor:\n");
for(i=0;i<5;i++)
{
    printf("%.2f\t",vetor[i]);//mostrando inicialização
    if(maior<vetor[i] || i==0)//buscando maior numero
    {
        maior=vetor[i];
    }
    soma+=vetor[i];//acumulando valores do vetor
    if(vetor[i]>5)
    {
        cont++;//quantos valores são maiores que 5
    }
}
//fim do for
printf("\n\nO maior valor do vetor \x82 %.2f", maior);
printf("\nA soma dos valores do vetor \x82 %.2f",soma);
printf("\nA media dos valores do vetor \x82 %.2f",soma/5);
printf("\nA qtde de valores >=5 \x82 %i",cont);
```



```
C:\Users\Talita\Documents\FACENS_Alunos\aula.exe
Valores armazenados no vetor:
7.00 3.50 10.00 5.00 8.00

O maior valor do vetor é 10.00
A soma dos valores do vetor é 33.50
A media dos valores do vetor é 6.70
A qtde de valores >=5 é 3
```

11. MATRIZES

Os elementos de um vetor podem ser de qualquer tipo, incluindo vetores, ou seja, em vez de guardar apenas um valor dentro de uma posição do vetor, pode-se armazenar outro conjunto de valores. Nesse caso, um vetor que possui um vetor de uma dimensão em cada elemento é chamado de vetor de duas dimensões.

Para facilitar a diferenciação um **vetor de duas ou mais dimensões** pode ser chamado de **matriz**. A quantidade do número de dimensões que um vetor pode ter não é determinado dentro da programação, mas sim pela capacidade do computador utilizado.

Cada elemento de uma matriz fica disposto sequencialmente na memória, como ocorre com os vetores, entretanto para facilitar o entendimento pode-se representá-la visualmente na forma matricial (linhas e colunas).

A sintaxe para declarar uma matriz é:

```
tipo_var nome_var[quantidade_linhas][quantidade_colunas];
```

Sendo assim, a quantidade representada no primeiro colchetes refere-se a quantidade de linhas da matriz e o segundo corresponde a quantidade de elementos que se pode armazenar em cada linha, portanto a quantidade de colunas.

Exemplo:

```
int numeros[3][4]; //a matriz contem 12 variáveis inteiras  
float avaliacoes[40][2]; // 40 alunos cada um com 2 notas
```

Para acessar as posições da matriz, utiliza-se o mesmo procedimento dos vetores, escreve-se o nome da matriz e dois pares de colchetes para indicar a posição de interesse. Índices de linhas e colunas começam em **zero**.

Exemplos:

```
int numeros[3][4], i, j; //i para linhas e j para colunas
for(i=0; i<3; i++) //for para percorrer as linhas
{
    for(j=0; j<4; j++) //for para percorrer as colunas
    {
        printf("Digite um numero [%i][%i]: ", i, j);
        scanf("%i", &numeros[i][j]);
    } //fim do for das colunas
} //fim do for das linhas
//Mesmo for aninhado para mostrar os valores da matriz
for(i=0; i<3; i++)
{
    for(j=0; j<4; j++)
    {
        printf("\nNumero[%i][%i]: %i", i, j, numeros[i][j]);
    }
}
```

Pode-se inicializar uma matriz também, logo na sua declaração, assim como é feito com vetores. A única ressalva seria em relação ao conjunto de inicialização que deverá ter as linhas da matriz como subconjuntos (chaves).

Exemplos:

```
int mat[3][2]={{1,2},{2,4},{4,16}}; //3 linhas e 2 colunas
float n[2][4]={{0,1,2,3},{1,2,4,5}}; //2 linhas e 4 colunas
```

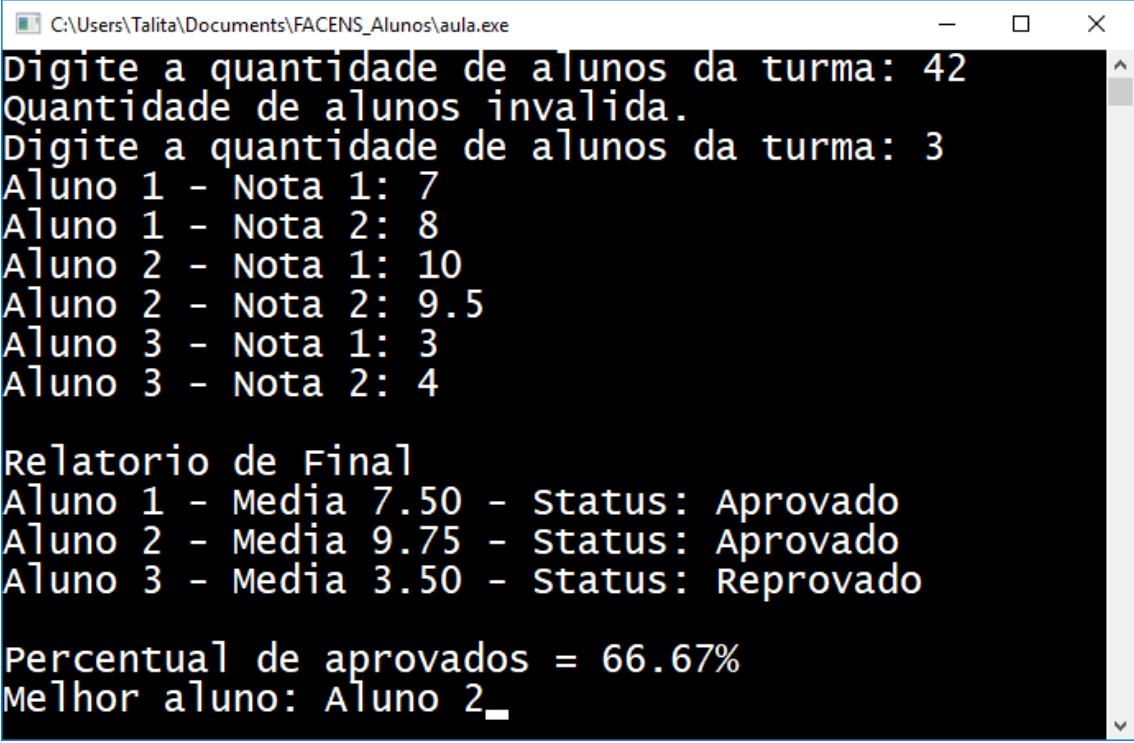
Abaixo segue um exemplo utilizando variáveis simples, um vetor e uma matriz, pois é perfeitamente possível trabalhar com todas as possibilidades em um mesmo algoritmo.

```
/*Calcular a média de provas de até 40 alunos e obter
algumas informações relevantes, tais como: situação de cada
aluno, porcentagem de aprovados e melhor aluno da turma*/
```

```
float avaliacoes[40][2], medias[40], soma=0, perc, maior;
int x;//variável x representa linha da matriz ou cada aluno
int y;//variável y representa linha da matriz ou cada aluno
int qtde, cont_aprovados=0,aux_aluno;
do
{
    printf("Digite a quantidade de alunos da turma: ");
    scanf("%i",&qtde);
    if(qtde<0 || qtde>40)
    {
        printf("Quantidade de alunos invalida.\n");
    }
}while(qtde<0 || qtde>40);
for(x=0;x<qtde;x++)
{
    for(y=0;y<2;y++)
    {
        printf("Aluno %i - Nota %i: ",x+1,y+1);
        scanf("%f",&avaliacoes[x][y]);
        soma+=avaliacoes[x][y];
    }
    medias[x]=soma/2;
    soma=0;//reinicializa soma
}
printf("\nRelatorio de Final\n");
for(x=0;x<qtde;x++)
{
    printf("Aluno %i - Media %.2f - ",x+1,medias[x]);
    if(medias[x]>=5.0)//Classificação do status
    {
        printf("Status: Aprovado\n");
        cont_aprovados++;
    }
}
```



```
    }  
    else  
    {  
        printf("Status: Reprovado\n");  
    }  
    if(medias[x]>maior || x==0)//Melhor aluno  
    {  
        maior=medias[x];  
        aux_aluno=x+1;  
    }  
}  
  
//um valor da operação deve ser float para resultado float  
perc=(cont_aprovados*100.00)/qtde;  
printf("\nPercentual de aprovados = %.2f%%",perc);  
printf("\nMelhor aluno: Aluno %i",aux_aluno);
```



```
C:\Users\Talita\Documents\FACENS_Alunos\aula.exe  
Digite a quantidade de alunos da turma: 42  
Quantidade de alunos invalida.  
Digite a quantidade de alunos da turma: 3  
Aluno 1 - Nota 1: 7  
Aluno 1 - Nota 2: 8  
Aluno 2 - Nota 1: 10  
Aluno 2 - Nota 2: 9.5  
Aluno 3 - Nota 1: 3  
Aluno 3 - Nota 2: 4  
  
Relatorio de Final  
Aluno 1 - Media 7.50 - status: Aprovado  
Aluno 2 - Media 9.75 - status: Aprovado  
Aluno 3 - Media 3.50 - status: Reprovado  
  
Percentual de aprovados = 66.67%  
Melhor aluno: Aluno 2_
```

APÊNDICE A: Acentuação na Linguagem C

Nos programas em C é possível inserir textos em português, no entanto se não houver o tratamento correto os acentos e cedilha não serão identificados pelo compilador. Sendo assim, ao executar um `printf()` ele não conseguirá exibir esses caracteres adequadamente, mostrando portanto algum caractere diferente do usado.

Exemplo: `printf("Aula de Programação");`

O texto no `printf()` acima, portanto, não será exibido da forma adequada.

Conforme já observado anteriormente é possível inserir códigos hexadecimais para inserir caracteres individualmente.

Exemplo: `printf("A aula \x82 sobre Linguagem C");`

A letra "é" na tabela ASCII tem o código 82, portanto, o `"\x82"` mostra o caractere com acento na tela.

No entanto quanto se quer escrever normalmente os textos na Linguagem C pode-se incluir a biblioteca `<locale.h>` e inserir na `main()` a função `"setlocale()"` para definir a localização (região/idioma) do programa. Assim os caracteres do idioma, acentuação, números e valores monetários e data e hora serão adaptados.

Exemplo:

```
#include<stdio.h>
#include<locale.h>
int main()
{
    setlocale(LC_ALL, "");
    printf("A aula é sobre Programação");
    return 0;
}
```

Nesse caso, a função `setlocale` recebe dois argumentos, o primeiro refere-se a todas as configuração de localização (`LC_ALL`), já o segundo argumento refere-se a linguagem utilizada. No caso ao se manter apenas as duas aspas duplas (sem espaço) a linguagem será a mesma do sistema operacional utilizado, ou seja, tornando o programa portátil. Outra possibilidade é inserir manualmente o palavra "Portuguese" no segundo argumento da `setlocale`.

Exemplo: `setlocale(LC_ALL, "Portuguese");`