



# **ESTRUTURA DE DADOS I (EDI)**

CIÊNCIA DA COMPUTAÇÃO

Amanda Danielle Lima de Oliveira Tameirão

Belo Horizonte

1 - 2018

## LISTA DE FIGURAS

FIGURA 1 – Evolução da linguagem C .....	6
FIGURA 2 – Características das variáveis .....	9
FIGURA 3 – Tipos de variáveis .....	11
FIGURA 4 – Declaração de variáveis locais .....	21
FIGURA 5 – Comando de Saída .....	24
FIGURA 6 – Resultado do comando de saída com texto .....	25
FIGURA 7 – Resultado do comando de saída com conteúdo de variável .....	25
FIGURA 8 – Resultado do comando de saída com texto e conteúdo de variável .....	26
FIGURA 9 – Comando de Entrada .....	27
FIGURA 10 – Estrutura de decisão simples .....	29
FIGURA 11 – Estrutura de decisão composta .....	31
FIGURA 12 – Estrutura de decisão aninhada .....	32
FIGURA 13 – Estrutura de decisão - Seletor .....	34
FIGURA 14 – Fluxograma da estrutura de repetição - <i>do while</i> .....	37
FIGURA 15 – Fluxograma da estrutura de repetição - <i>while</i> .....	38
FIGURA 16 – Fluxograma da estrutura de repetição - <i>while</i> .....	40
FIGURA 17 – Armazenamento de uma string (vetor de char) .....	48
FIGURA 18 – Armazenamento de memória .....	54
FIGURA 19 – Armazenamento de ponteiros .....	56
FIGURA 20 – Função com passagem de parâmetro .....	69
FIGURA 21 – Função sem passagem de parâmetro .....	70

## LISTA DE QUADROS

QUADRO 1 – Tipos de variáveis .....	12
QUADRO 2 – Operadores matemáticos .....	16
QUADRO 3 – Operadores relacionais .....	17
QUADRO 4 – Operadores lógicos .....	18
QUADRO 5 – Tabela verdade do operador E .....	19
QUADRO 6 – Tabela verdade do operador OU .....	19
QUADRO 7 – Tabela verdade do operador NÃO .....	20
QUADRO 8 – Simplificando atribuições .....	21
QUADRO 9 – String de controle dos tipos de variáveis.....	26
QUADRO 10 – Caracteres especiais para impressão .....	26
QUADRO 11 – Endereçamento de vetores .....	57
QUADRO 12 – Endereçamento de vetores .....	60
QUADRO 13 – Tipos básicos para retorno de funções .....	68
QUADRO 14 – Passagem de parâmetro por valor – Análise de código .....	74
QUADRO 15 – Passagem de parâmetro por referência – Análise de código .....	75
QUADRO 16 – Declaração de novos tipos .....	80

## **LISTA DE ABREVIATURAS E SIGLAS**

EDI – Estrutura de Dados I

LTPI – Linguagens e Técnicas de Programação I

LTPII – Linguagens e Técnicas de Programação II

SO – Sistema Operacional

## SUMÁRIO

<b>1</b>	<b>BOAS VINDAS .....</b>	<b>5</b>
<b>2</b>	<b>CONCEITOS BÁSICOS .....</b>	<b>6</b>
2.1	A linguagem C .....	6
2.2	Estrutura Básica .....	6
2.3	Diretivas de Compilação .....	7
2.3.1	<i>A diretiva include</i> .....	7
2.4	Função principal .....	8
<b>3</b>	<b>VARIÁVEIS .....</b>	<b>9</b>
3.1	Características importantes .....	9
3.1.1	<i>Nome</i> .....	10
3.1.1.1	<u>Regras de criação de nomes</u> .....	10
3.2	Tipos de variáveis .....	11
3.2.1	<i>Modificadores de tipos de variáveis</i> .....	12
3.3	Declaração .....	13
3.4	Inicialização .....	14
3.5	Escopo .....	14
3.5.1	<i>Escopo - Global</i> .....	14
3.5.2	<i>Escopo - Local</i> .....	15
<b>4</b>	<b>OPERADORES E EXPRESSÕES .....</b>	<b>16</b>
4.1	Operadores aritméticos .....	16
4.2	Operadores relacionais .....	17
4.3	Operadores lógicos .....	18
4.3.1	<i>Operador lógico E - &amp;&amp;</i> .....	18
4.3.2	<i>Operador lógico OU -   </i> .....	19
4.3.3	<i>Operador lógico NÃO - !</i> .....	20
4.4	Operador de atribuição .....	20
4.4.1	<i>Simplificando expressões</i> .....	21
	<b>EXERCÍCIOS .....</b>	<b>22</b>

<b>5</b>	<b>COMANDOS BÁSICOS .....</b>	<b>24</b>
5.1	Comando de saída .....	24
5.2	Comando de entrada .....	26
	<i>EXERCÍCIOS</i> .....	28
<b>6</b>	<b>CONTROLADORES DE FLUXO .....</b>	<b>29</b>
6.1	Estruturas de Decisão .....	29
6.1.1	<i>Comando Se - if</i> .....	29
6.1.1.1	<u>Estrutura de decisão simples</u> .....	29
6.1.1.2	<u>Estrutura de decisão composta</u> .....	31
6.1.1.3	<u>Estrutura de decisão aninhada</u> .....	32
6.1.1.4	<u>Switch</u> .....	34
6.1.1.5	<u>Operador ternário - ?</u> .....	36
6.2	Estruturas de Repetição .....	37
6.2.1	<i>Comando com condição no final - do while</i> .....	37
6.2.2	<i>Comando com condição no início - while</i> .....	38
6.2.3	<i>Comando com contador de iterações - for</i> .....	40
6.3	Comandos auxiliares .....	41
6.3.1	<i>break</i> .....	41
6.3.2	<i>continue</i> .....	43
	<i>EXERCÍCIOS</i> .....	44
<b>7</b>	<b>VARIÁVEL COMPOSTA HOMOGÊNEA .....</b>	<b>45</b>
7.1	Unidimensionais - Vetores .....	45
7.2	Bidimensionais - Matrizes .....	46
7.3	Não dimensionadas .....	46
<b>8</b>	<b>MANIPULAÇÃO DE STRINGS .....</b>	<b>47</b>
8.1	Funções da biblioteca <code>string.h</code> .....	48
8.1.1	<i>gets</i> .....	48
8.1.2	<i>strlen</i> .....	48
8.1.3	<i>strcpy</i> .....	49
8.1.4	<i>strncpy</i> .....	50
8.1.5	<i>strcat</i> .....	50

8.1.6	<i>strncat</i> .....	51
8.1.7	<i>strcmp</i> .....	52
	<b>EXERCÍCIOS</b> .....	53
<b>9</b>	<b>PONTEIROS E ALOCAÇÃO DINÂMICA</b> .....	<b>54</b>
9.1	Conceito de ponteiro .....	54
9.1.1	<i>Vantagens de utilizar um ponteiro</i> .....	54
9.1.2	<i>Como declarar um ponteiro</i> .....	54
9.2	Operador & .....	55
9.3	Operador * .....	56
9.4	Vetores x Ponteiros .....	57
9.4.1	<i>Atribuição em vetores</i> .....	58
9.4.2	<i>Manipulação de vetores</i> .....	59
	<b>EXERCÍCIOS</b> .....	60
<b>10</b>	<b>ALOCAÇÃO DINÂMICA</b> .....	<b>62</b>
10.1	Divisão da memória .....	62
10.1.1	<i>Segmento de código</i> .....	63
10.1.2	<i>Segmento de dados</i> .....	63
10.1.3	<i>Stack</i> .....	63
10.1.4	<i>Heap</i> .....	63
10.2	Funções para controle da memória Heap .....	63
10.2.1	<i>Função malloc</i> .....	63
10.2.2	<i>Função free</i> .....	64
10.2.3	<i>Função calloc</i> .....	65
10.2.4	<i>Função realloc</i> .....	66
	<b>EXERCÍCIOS</b> .....	66
<b>11</b>	<b>FUNÇÕES</b> .....	<b>67</b>
11.1	Estrutura básica de uma função .....	67
11.1.1	<i>Tipo de retorno</i> .....	68
11.1.2	<i>Nome</i> .....	70
11.1.3	<i>Protótipo da função</i> .....	70
11.1.4	<i>Comandos da função</i> .....	72

<i>11.1.5 Passagem de parâmetro</i> .....	72
<i>11.1.5.1 Como efetuar passagem de parâmetro</i> .....	72
<i>11.1.5.2 Passagem de parâmetro por valor</i> .....	73
<i>11.1.5.3 Passagem de parâmetro por referência</i> .....	74
<i>11.2 Passagem de vetores por parâmetro</i> .....	76
<i>11.2.1 Passagem de vetores por parâmetro – Aritmética de ponteiros</i> .....	77
<i>11.2.2 Passagem de vetores por parâmetro – Índice de vetor</i> .....	77
<b>12 DECLARAÇÃO DE NOVOS TIPOS</b> .....	<b>79</b>
<b>12.1 Typedef</b> .....	79
<b>12.2 Enum - Enumeration</b> .....	80
<b>12.3 Struct - Estruturas (Registros)</b> .....	82
<b>12.4 Union - União</b> .....	84
<b>EXERCÍCIOS</b> .....	84
<b>REFERÊNCIAS</b> .....	<b>86</b>



## 1 BOAS VINDAS

Olá aluno(a), a partir de agora iniciaremos a disciplina de Estrutura de Dados I (EDI).

Para termos sucesso nesta disciplina, utilizaremos muitos conceitos aprendidos em disciplinas anteriores, como por exemplo, Arquitetura, Linguagens e Técnicas de Programação I (LTPI) e alguns conceitos de Linguagens e Técnicas de Programação II (LTPII).

Para me auxiliar na escrita desta apostila utilizei alguns livros, apostilas e sites para pesquisa. Observe na lista abaixo, para que você também possa acessá-los:

- 1 - Notas de aula do professor Flavio Lapper
- 2 - Livro da autora Ana Fernanda Ascencio (ASCENCIO; CAMPOS, 2012)
- 3 - Livro do autor Adam Drozdek (DROZDEK, 2002)
- 4 - Livro dos autores André Forbellone e Henri Eberspacher (FORBELLONE; EBERSPACHER, 2005)
- 5 - Livro do autor Bruno Preiss (PREISS, 2000)
- 6 - Livro do autor Celso Moraes (MORAES, 2001)
- 7 - Livro dos autores Michael Goodrich e Roberto Tamassia (GOODRICH; TAMASSIA, 2007)
- 8 - Livro do autor Nivio Ziviani (ZIVIANI, 2004)
- 9 - Livro da autora Viviane Mizrahi (MIZRAHI, 2008)

Se for necessário, acesse este material para tirar dúvidas ou fazer alguma consulta. As referências estão no final da apostila.

Para finalizar, reintero que é com grande alegria que inicio esta caminhada ao seu lado, espero que possamos ensinar e aprender muito juntos.

Lembre-se que sempre estarei aqui para auxiliá-lo.

Abraços e bons estudos,

Prof<sup>a</sup> Amanda Danielle Lima de Oliveira Tameirão

## 2 CONCEITOS BÁSICOS

Para esta disciplina, aprenderemos as estruturas utilizando a linguagem C, mas, gostaria de avisá-lo que, mais que desenvolver programas, precisamos compreender a execução e lógica envolvida nas soluções. Portanto, por diversas vezes, você precisará "elevar" seus pensamentos além da linguagem ("pensar fora da caixinha", como dizem por aí).

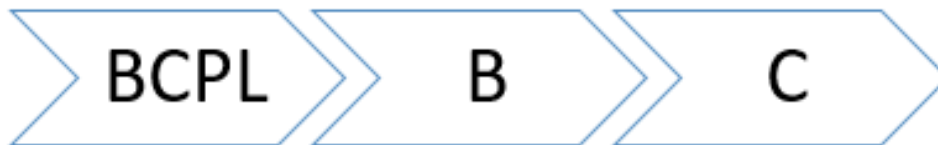
### 2.1 A linguagem C

A linguagem C foi criada por Dennis Ritchie, Martin Richards e Ken Thompson, na década de 70.

É uma linguagem baseada em outras, que busca sua força na construção modular de seus programas.

A Figura abaixo exibe a evolução da linguagem.

**Figura 1 – Evolução da linguagem C**



**Fonte: Elaborado pela autora**

É uma linguagem poderosa, responsável pelo desenvolvimento de diversos programas conhecidos, entre eles sistemas operacionais, planilhas eletrônicas, processadores de textos, etc.

A linguagem é *case sensitive*, ou seja, diferencia letras maiúsculas de letras minúsculas, neste caso, as letras "A" e "a" são diferentes.

### 2.2 Estrutura Básica

Para escrever um programa na linguagem C, é preciso "obedecer" à sua estrutura básica.

A estrutura correta é:

- \* Declarações globais - Que permitem que todas as funções os acessem.  
Os exemplos são, variáveis globais, diretivas de compilação, etc.
- \* Funções do desenvolvedor - São as funções desenvolvidas por ele, com o intuito de modularizar o programa.
- \* Função principal - Responsável por iniciar a execução do programa.

```

1  #include <stdio.h> //Diretivas de compilação
2
3  void mensagem(){ //Função do desenvolvedor
4      printf ("Exibir mensagem");
5  }
6
7  int main(){ //Função principal
8      printf ("Meu programa começa aqui.");
9      return 0; //Retorno da função
10 }

```

## 2.3 Diretivas de Compilação

Uma diretiva de compilação é uma instrução enviada ao pré-processador (programa que efetua modificações e execuções necessárias à execução).

Toda diretiva inicia-se com um “#”, e como não é um comando da linguagem, não precisa finalizar com ; (ponto e vírgula). Além disto, seu texto deve ser escrito em uma única linha.

Exemplos de diretivas de compilação:

- include: inclui o arquivo indicado
- define: define uma macro
- if, elif, else, endif: compilação condicional

### 2.3.1 A diretiva *include*

Ao usarmos a diretiva *include*, estamos incluindo o arquivo solicitado em nosso código fonte. Antes da compilação, o pré-processador inclui o arquivo indicado e só depois compila.

A sintaxe correta ao declarar uma diretiva é:

- \* Para utilizar arquivos já pertencentes a linguagem e que estão na pasta *include*, utilize a seguinte sintaxe

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

- \* Para utilizar uma biblioteca que possua funções criadas pelo próprio desenvolvedor, utilize a seguinte sintaxe

```
#include "nome do arquivo"
```

Neste caso ele buscará até 10 níveis internos.

Arquivos comuns em nossa utilização:

`#include <stdio.h>` - Funções de Entrada e Saída

`#include <stdlib.h>` - Biblioteca padrão da linguagem C

`#include <math.h>` - Funções matemáticas

`#include <string.h>` - Manipulação de Strings

**Atenção:** Sei que a explicação está pouco detalhada, mas, é o que você precisa saber neste momento. Teremos um conteúdo mais detalhado sobre isto em outro capítulo.

## 2.4 Função principal

A execução de um programa em C inicia-se através da função principal, denominada *main*.

É a partir dela que ele invoca as outras funções, portanto, todo o programa que executa na linguagem C inicia-se na função *main*.

Quem a executa é o pré-processamento, a função retorna um valor inteiro a ele, indicando se foi executada com sucesso ou se houve algum erro.

```
1  int main(){ //Cabeçalho da função principal
2      printf ("Meu programa começa aqui."); //Bloco de comandos
3      return 0; //Retorno da função
4  }
```

No próximo Capítulo falaremos sobre os espaços de memória que são reservados para utilizarmos os programas, acredito que você já sabe que falaremos sobre as **variáveis**.

### 3 VARIÁVEIS

A memória física de um computador é utilizada para efetuar os armazenamentos que ele necessita em diversos programas e dispositivos, para isto, é necessário reservar um espaço específico. A este espaço damos o nome de variáveis.

Por exemplo:

- \* Se precisar armazenar o nome de alguém, criará uma variável para guardá-lo.
- \* Se precisar armazenar a idade, criará uma variável para guardá-la.
- \* E isto deverá ocorrer para TODAS as informações que você deseja armazenar em seu programa.

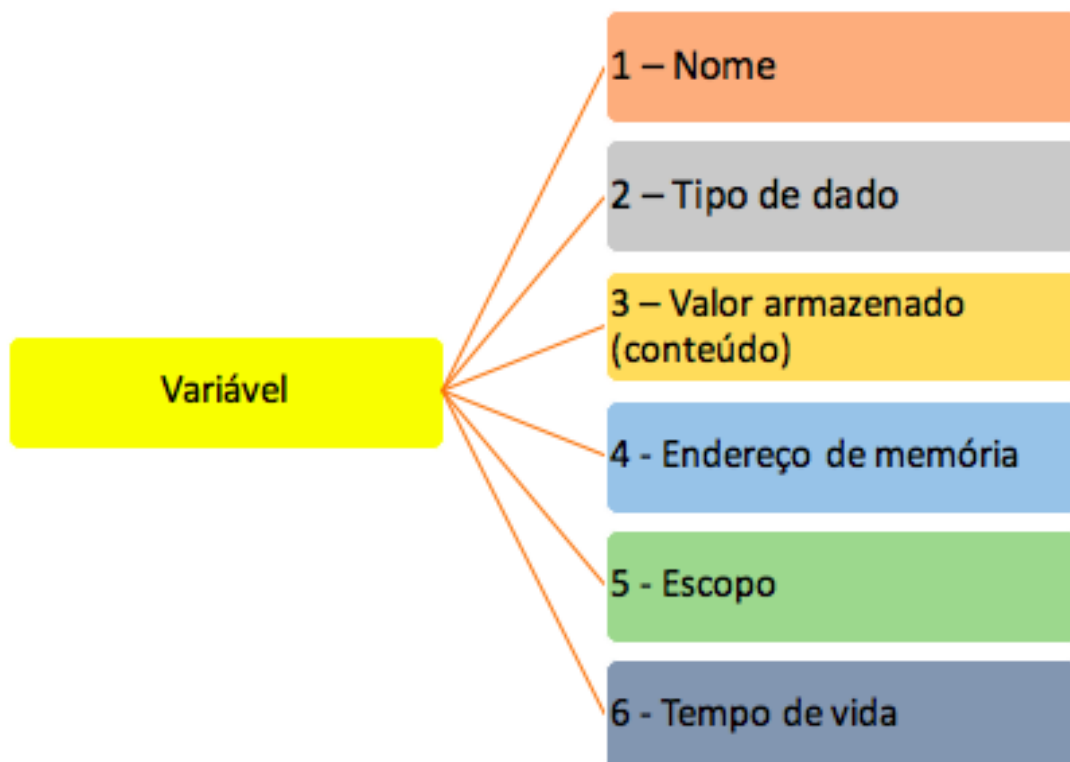
Agora descobriremos quais as características de uma variável.

#### 3.1 Características importantes

Para pensarmos em estruturas de dados, é importante saber que estas variáveis possuem características relevantes para a utilização.

Vamos observar cada uma:

**Figura 2 – Características das variáveis**



**Fonte:** Elaborado pela autora

Agora detalharemos cada uma dessas características.

### 3.1.1 *Nome*

Ao indicarmos um nome a uma variável, estamos a identificando dentro do programa.

Mas atenção, o nome de uma variável deve seguir algumas regras, para que o programa funcione corretamente.

Observe:

#### 3.1.1.1 Regras de criação de nomes

É importante observar que algumas regras fazem parte da sintaxe da linguagem, e outras são convenções, que no mercado de trabalho dão agilidade a escrita e manutenção dos programas.

Ambas devem ser respeitadas e seguidas! Lembre-se que você não escreve um programa apenas para você ler, outras pessoas utilizarão e deverão compreender seu código.

Vamos às regras:

- \* Começar com letras ou underline ( \_ )

Sempre que criar uma variável é imprescindível que ela comece com letras ou underline ( \_ ), assim ela será aceita pelo programa.

**Nunca** comece uma variável com números ou caracteres especiais. Isto não é permitido!

A partir da segunda posição pode-se utilizar letra, número ou underline ( \_ ).

Exemplo: idade, nome1200, salario\_final, somaTotal

- \* Utilizar letras minúsculas

As variáveis são escritas, preferencialmente, com letras minúsculas. Se houver duas ou mais palavras, cada nova palavra deverá iniciar com letra maiúscula (este item é conhecido como *camelCase*).

Exemplo: maiorIdade, salarioFinal, media\_geral, salario, idade2

- \* Nomes simples e descritivos

É importante que o nome de uma variável seja de fácil compreensão para você, que está escrevendo o programa, e para as pessoas que darão manutenção no código.

Exemplo: uma variável que possui o nome "idade" armazenará a idade de algo, ou alguém, dentro do código. É fácil concluir isto.

\* Não utilize palavras reservadas

Algumas palavras pertencem a estrutura da linguagem, e não podem ser utilizadas para nomes de variáveis.

Exemplo else, char, int, etc.

\* Linguagem *Case Sensitive*

Ao escrever um código, é importante observar se a linguagem é *case sensitive*, ou seja, se diferencia letras maiúsculas de letras minúsculas.

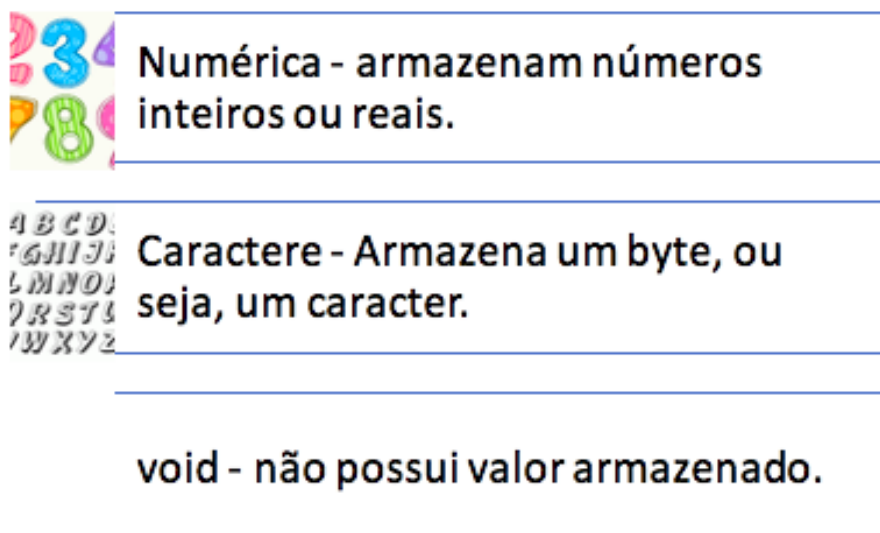
Em linguagens *case sensitive*, uma variável com o nome de **idade** é diferente da variável de nome **IDADE**, e também diferente da variável **Idade**, etc.

### 3.2 Tipos de variáveis

Definir o tipo de uma variável é o mesmo que definir o que ela poderá armazenar, ou seja, qual o conteúdo você poderá guardar dentro daquela variável.

Você pode declarar três tipos possíveis de variáveis:

**Figura 3 – Tipos de variáveis**



**Fonte:** Elaborado pela autora

Cada tipo de variável efetua um armazenamento distinto de *bytes* na memória. Para descobrir a quantidade, utilize o comando *sizeof* da linguagem, utilizando a seguinte sintaxe:

```

1  #include <stdio.h>
2
3  int main(){
4      printf("O armazenamento de uma variável do tipo int é %i bytes.", sizeof(int));
6      return 0;
7  }

```

Veja no quadro abaixo os tipos de variáveis possíveis para declaração na linguagem C.

**Quadro 1 – Tipos de variáveis**

Tipo		Número de bytes	Início	Fim
Caractere	<i>char</i>	1	-128	127
	<i>unsigned char</i>	1	0	255
Número inteiro	<i>short int</i>	2	-32.765	32.767
	<i>unsigned short int</i>	2	0	65.535
	<i>int</i>	4	-2.147.483.648	2.147.483.647
	<i>unsigned int</i>	4	0	4.294.967.295
	<i>long int</i>	4	-2.147.483.648	2.147.483.647
	<i>unsigned long int</i>	4	0	4.294.967.295
Número real	<i>float</i>	4	$3,4 \times 10^{-38}$	$3,4 \times 10^{38}$
	<i>double</i>	8	$1,7 \times 10^{-308}$	$1,7 \times 10^{308}$
	<i>long double</i>	8	$3,4 \times 10^{-4932}$	$3,4 \times 10^{4932}$
void	<i>void</i>	Nenhum valor		

**Fonte:** Elaborado pela autora

### 3.2.1 Modificadores de tipos de variáveis

Um modificador efetua uma alteração no tipo original da variável, permitindo que seu armazenamento de memória seja maior ou menor, dependendo de sua aplicação.

Os modificadores possíveis são:

**short** - indica que a variável será menor que seu valor original.

Exemplo: Uma variável inteira armazena 4 *bytes*, se ela for *short int* armazenará 2 *bytes*.

**long** - indica que a variável será maior que seu valor original.

Exemplo: Uma variável inteira armazena 4 *bytes*, se ela for *long int* armazenará 8 *bytes*.

**unsigned** - indica que a variável não armazenará valores negativos.

Exemplo: Uma variável *char* armazena valores de -128 a 127, se ela for *unsigned char* armazenará de 0 até 255.

Observações:



- \* Toda variável é naturalmente do tipo *signed*, pois sempre permite valores negativos e positivos.
- \* Ao tipo *float* não se pode aplicar nenhum modificador
- \* Ao tipo *double* pode-se aplicar apenas o modificador *long*
- \* Todos os modificadores podem ser aplicados ao tipo *int*

Cada compilador é livre para escolher tamanhos adequados para o seu próprio *hardware* com algumas restrições:

Os *short's* e *int's* devem ocupar pelo menos 16 *bits*

Os *longs* pelo menos 32 *bits*

O *short* não pode ser maior que *int*

O *int* não pode ser maior que *long*

### 3.3 Declaração

Você já sabe que declarar variável é o mesmo que "separar" um espaço de memória para armazenar informações que serão úteis aos programas.

Para declararmos uma variável, em C, basta utilizarmos a seguinte sintaxe:

<tipo da variável> <nome da variável>;

Onde o tipo da variável, segue a tabela já citada (Tabela 1), e o nome segue as descrições já ensinadas anteriormente.

Veja a seguir alguns exemplos destas declarações:

```

1   int main(){
2       int idade;
3       float somaTotal, total;
4       double salario;
5       char letra, frase[30];
6       return 0;
7   }
```

Seguindo os exemplos acima, podemos utilizar qualquer tipo de variável para declaração.

Se as variáveis forem do mesmo tipo, pode-se declará-las apenas separando por vírgulas, ao finalizar, inclua ponto e vírgula.

### 3.4 Inicialização

Inicializar uma variável corresponde a dar um valor para que ela comece o programa, ou seja, incluir um conteúdo nesta variável.

Considerando as necessidades do programa, uma variável deve ser inicializada a partir de duas regras importantes:

1ª Regra - Toda variável que recebe ela mesma precisa ter um valor inicial para começar a lógica.

2ª Regra - Toda variável que for testada antes de receber um valor precisa ser inicializada para que o teste tenha resultado.

Nos demais casos, você pode inserir valor a uma variável sempre que precisar, não existe regra específica. Para isto utilize o operador de atribuição, que, na linguagem C, corresponde a um símbolo de igualdade (=).

Uma variável pode ser inicializada em dois momentos do programa:

1º - Ao declará-la -

```
int idade = 12;
```

2º - Durante as linhas de código, após a declaração -

```
int idade;  
  
idade = 12;
```

### 3.5 Escopo

Definir o escopo de uma variável, indica saber onde ela será válida, ou seja, onde ela será visível para a execução do programa.

Atenção: Os parâmetros de uma função são considerados variáveis locais, pois possuem as mesmas características.

#### 3.5.1 *Escopo - Global*

Uma variável é considerada global quando declarada fora de todas as funções.

Ela será visível por todo o programa, ou seja, ela não "pertence" a nenhuma função específica, qualquer uma poderá acessar, e alterar, seu conteúdo.

```

1  #include <stdio.h>
2  int valor;
3  void soma(){
4      printf("%d", valor + valor);
5  }
6  int main(){
7      scanf("%d", &valor);
8      soma();
9      return 0;
10 }
```

### 3.5.2 Escopo - Local

Uma variável é considerada local quando declarada dentro de uma função, e só poderá ser visível, e alterada, pela função em que foi declarada.

```

1  #include <stdio.h>
2  void soma(){
3      int valor;
4      scanf("%d", &valor);
5      printf("%d", valor + valor);
6  }
7  int main(){
8      int valor;
9      scanf("%d", &valor);
10     soma();
11     return 0;
12 }
```

No código acima, as variáveis das linhas 3 e 8, apesar de ter o mesmo nome, são variáveis distintas, ocupam espaços de memória diferentes.

As regras que regem onde uma variável é válida são:

- \* Duas variáveis globais não podem ter o mesmo nome.
- \* O mesmo vale para duas variáveis locais, de uma mesma função.
- \* Duas variáveis locais, de funções diferentes, podem ter o mesmo nome sem perigo de conflito. Mas atenção, não serão a mesma variável.

Bem, por aqui finalizamos todas as explicações sobre as variáveis.

## 4 OPERADORES E EXPRESSÕES

Em linguagens de programação, os operadores permitem a execução de alguma expressão ou avaliação lógica. Onde, uma expressão corresponde a alguma "frase" que necessita de interpretação, ou execução.

É comum que o código aguarde o resultado da expressão, aplicada com operadores, para continuar a executar as próximas linhas do programa.

Veja a seguir os tipos de operadores disponíveis para utilização.

### 4.1 Operadores aritméticos

Os operadores aritméticos permitem execuções matemáticas ao código. Para isto, é necessário aplicar algum dos seguintes itens:

**Quadro 2 – Operadores matemáticos**

Operador	Descrição	Exemplos
+	Soma	1.5 + 45.9 12 + 56 -23.8 + x
-	Subtração	y - w 1.45 - 5 -x - 59
*	Multiplicação	y * w 1.8 * 55 -x * 5.4
/	Divisão (Em C, a divisão de inteiros retorna um valor inteiro. Para obter valores decimais é necessário fazer um <i>typecasting</i> )	18 / 65 12.12 / x y / 5
%	Resto da divisão inteira (Esse cálculo deverá ser feito por duas variáveis, ou valores, inteiros. O resultado sempre será um valor inteiro)	15 % 4 x % 65 y % w soma % contador
++	incremento	cont++ incrementa 1 na variável cont cont++ é o mesmo que cont = cont + 1 ou cont += 1
--	decremento	cont-- decrementa 1 na variável cont cont-- é o mesmo que cont = cont - 1 ou cont -= 1

**Fonte: Elaborado pela autora**

Observação: Para o operador de divisão inteira, no cálculo 15 % 4, sabe-se que o resto de 15 dividido por 4 é 3, então, esse será o resultado desta expressão (15 / 4 = 3 e tem resto 3).

## 4.2 Operadores relacionais

Ao utilizarmos um operador relacional, estamos avaliando uma relação entre **DUAS** expressões. Se necessário, cada expressão poderá utilizar operadores aritméticos, sendo que as expressões aritméticas serão solucionadas antes das expressões relacionais.

**Quadro 3 – Operadores relacionais**

Operador	Descrição	Exemplos
==	igualdade	$x - y == 12 * a$ $12 == 56.9$ $-23.8 == x + y$
!=	Diferença	$cont != 12$ $1 != 5.9$ $-x + 23 != y / 5$
>	Maior	$soma * 10 > x / 2$ $1.8 > 0.5$ $-x > 5.4 / 2$
>=	Maior ou igual	$y \% 2 >= 54 \% 3$ $189.8 >= 65$ $y >= x * 2$
<	Menor	$idade < menorIdade$ $0.8 < 6.5$ $x - 34 < 23 * 54$
<=	Menor ou igual	$y - w <= cont * 72$ $189.8 <= x$ $y <= 72$

**Fonte: Elaborado pela autora**

É importante saber que **TODA** avaliação relacional terá seu resultado como verdadeiro (*true*) ou falso (*false*).

Veja os exemplos abaixo:

Para ambos, considere as seguintes variáveis

`int soma = 235,`

`cont = 0;`

Primeiro exemplo:

`soma >= cont * 10`

`235 >= 0 * 10`

`235 >= 0`

*true*

Segundo exemplo:

`soma % 10 == cont++`

`235 % 10 == 1`

`5 == 1`

`false`

Observações -

`soma % 10` - o resto da divisão de 235 por 10 é igual a 5

`cont++` - considerando que a variável `cont` é zero, zero + 1 é igual a 1

O processador sempre executará esta avaliação, "pegará" o resultado, como verdadeiro ou falso, e só depois, dependendo do comando utilizado, ele irá decidir o que fazer.

### 4.3 Operadores lógicos

Os operadores lógicos permitem a execução de mais de um relacionamento. Assim, sua avaliação será ampliada para diversas análises.

Vamos verificar quais são esses operadores:

**Quadro 4 – Operadores lógicos**

Operador	Descrição
<code>&amp;&amp;</code>	E (conjunção)
<code>  </code>	OU (disjunção)
<code>!</code>	NÃO (negação)

**Fonte: Elaborado pela autora**

#### 4.3.1 Operador lógico E - `&&`

O operador lógico E, efetua a junção das expressões avaliadas. Estas expressões devem ser lógicas, ou seja, devem ter resultados verdadeiro (*true*) ou falso (*false*). Em Java, utilizamos os caracteres `&&` para representar este operador.

Sempre que precisar utilizar operador lógico E, você deve considerar que todas as expressões devem ser verdadeiras para que o resultado final seja verdadeiro, caso contrário, o resultado será falso.

Portanto, se tivermos 1000 expressões, onde 999 são verdadeiras e 1 é falsa, o resultado final será falso. É imprescindível que **TODAS** as expressões sejam verdadeiras para que o resultado final seja verdadeiro.

A tabela verdade do operador E é:

**Quadro 5 – Tabela verdade do operador E**

Expressão 1		Expressão 2	Resultado Final
Verdadeiro	&&	Verdadeiro	Verdadeiro
Verdadeiro	&&	Falso	Falso
Falso	&&	Verdadeiro	Falso
Falso	&&	Falso	Falso

**Fonte: Elaborado pela autora**

No exemplo foram utilizadas apenas duas expressões, mas pode-se utilizar quantas forem necessárias, desde que sejam "ligadas" por um operador lógico.

#### 4.3.2 Operador lógico OU - ||

O operador lógico OU, efetua a disjunção (separação) das expressões avaliadas. Estas expressões devem ser lógicas, ou seja, devem ter resultados verdadeiro (*true*) ou falso (*false*). Em Java, utilizamos os caracteres || para representar este operador.

Sempre que precisar utilizar o operador lógico OU, você deve considerar que **NÃO** será necessário que todas as expressões sejam verdadeiras, basta que uma seja verdadeira para que o resultado final seja verdadeiro. O resultado final só será falso se **TODAS** as expressões forem falsas.

Portanto, se tivermos 1000 expressões, onde 999 são falsas e 1 é verdadeira, o resultado final será verdadeiro. É imprescindível que **TODAS** as expressões sejam falsas para que o resultado final seja falso.

No operador OU pode ser que uma ou mais condições sejam atendidas, isso significa que todas também podem ser atendidas.

A tabela verdade do operador OU é:

**Quadro 6 – Tabela verdade do operador OU**

Expressão 1		Expressão 2	Resultado Final
Verdadeiro		Verdadeiro	Verdadeiro
Verdadeiro		Falso	Verdadeiro
Falso		Verdadeiro	Verdadeiro
Falso		Falso	Falso

**Fonte: Elaborado pela autora**

No exemplo foram utilizadas apenas duas expressões, mas pode-se utilizar quantas forem necessárias, desde que sejam "ligadas" por um operador lógico.

#### 4.3.3 Operador lógico NÃO - !

O operador lógico NÃO efetua uma negação na expressão avaliada. Isso significa que irá inverter o valor original da expressão negada. Em Java, utilizamos o caractere ! para representar este operador.

Sempre que precisar utilizar o operador lógico NÃO, a ideia é que o resultado atual será invertido, ou seja, se o resultado for verdadeiro, o resultado final é igual a falso. Afinal, **NÃO** verdadeiro será igual a falso (e vice-versa).

A tabela verdade do operador NÃO é:

**Quadro 7 – Tabela verdade do operador NÃO**

Expressão	Resultado Final
! Verdadeiro	Falso
! Falso	Verdadeiro

**Fonte: Elaborado pela autora**

#### 4.4 Operador de atribuição

Atribuir valor a uma variável corresponde a colocar valor dentro do espaço de memória em que ela está.

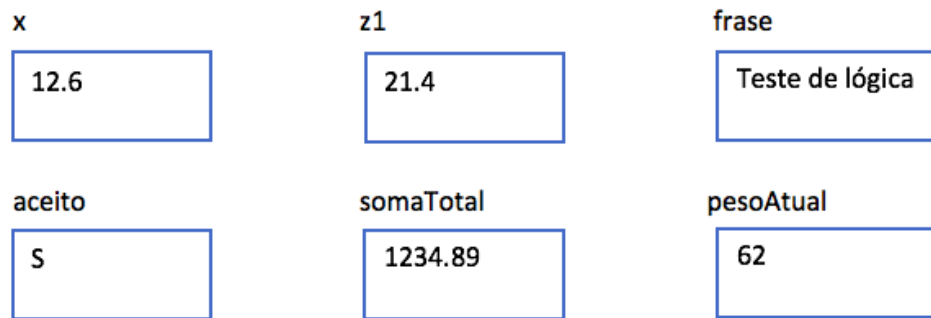
Na linguagem C, utilizamos o caracter = (igual) para efetuar esta atribuição, ou seja, para incluir valor dentro de alguma variável.

<nome\_variável> = <valor\_atribuído>;

Exemplos:

```
x = 12.6;
z1 = 34 - x;
frase = "Teste de lógica";
aceito = 'S';
somaTotal = 1234.89;
pesoAtual = 62;
```



**Figura 4 – Declaração de variáveis locais**

**Fonte:** Elaborado pela autora

Quanto a utilização, fique atento:

- \* Os valores reais devem ser escritos com ponto;
- \* Os valores atribuídos sempre devem atender ao tipo da variável, ou seja, não se deve colocar um conteúdo alfanumérico dentro de uma variável numérica, etc;
- \* O operador de atribuição (=) é diferente do operador relacional de igualdade (==).

#### 4.4.1 Simplificando expressões

Em códigos de programação, é comum que uma variável precise armazenar seu próprio conteúdo, e "somar" a outros, neste caso, dizemos que a variável recebe ela mesma, ou seja, mantém seu próprio valor para o cálculo.

Toda vez que uma variável precisar receber ela mesma, uma forma prática, e muito utilizada para simplificar as expressões, é utilizar o operador de atribuição da seguinte forma:

Vamos imaginar que iremos efetuar os seguintes cálculos

**Quadro 8 – Simplificando atribuições**

float      somaIdade = 0, idade = 26;      //As inicializações são apenas para exemplificar	
Código comum	Código simplificado
somaIdade = somaIdade + idade;	somaIdade += idade;
somaIdade = somaIdade * 10;	somaIdade *= 10;
somaIdade = somaIdade / 150;	somaIdade /= 150;
somaIdade = somaIdade % 5;	somaIdade %= 5;

**Fonte:** Elaborado pela autora

## EXERCÍCIOS

1. Considerando que  $A = 12$ ,  $B = 45$  e  $C = 0$ , leia as expressões aritméticas a seguir e informe o resultado de cada uma:

- a. \_\_\_\_\_  $A * B + C$
- b. \_\_\_\_\_  $A \% B$
- c. \_\_\_\_\_  $C + B - A$
- d. \_\_\_\_\_  $B / A$
- e. \_\_\_\_\_  $C / 2 - (B + 4) - A * 6$
- f. \_\_\_\_\_  $(A - B) * C / 3 + 4$
- g. \_\_\_\_\_  $A * 3 - B * 5 + A \% 3$
- h. \_\_\_\_\_  $A \% 2 + B \% 5 + C \% 9$

2. Considerando que  $A = 2$ ,  $B = 4$  e  $C = 5$ , leia as expressões relacionais a seguir e informe o resultado de cada uma. Lembre-se que estas só terão resultados verdadeiros ou falsos:

- a. \_\_\_\_\_  $A != B$
- b. \_\_\_\_\_  $A * B == B / C$
- c. \_\_\_\_\_  $A / 2 <= B + 5$
- d. \_\_\_\_\_  $A - 12 / B + 45 < C + 9 * 3$
- e. \_\_\_\_\_  $A \% 2 > B \% 2$
- f. \_\_\_\_\_  $-A >= -B + C$
- g. \_\_\_\_\_  $A / 3 * 2 - C < 12$
- h. \_\_\_\_\_  $A / 3 * 2 - C >= A \% 10 - C * B$

6. Considerando que  $A = 78$ ,  $B = 64$  e  $C = 32$ , leia as expressões lógicas a seguir e informe o resultado de cada uma. Lembre-se que estas só terão resultados verdadeiros ou falsos:

- a. \_\_\_\_\_  $A == B \ \&\& \ A == C$
- b. \_\_\_\_\_  $A != B \ || \ A == C$
- c. \_\_\_\_\_  $12 >= C - 45 \ \&\& \ A > B$
- d. \_\_\_\_\_  $(C - 12) \% 4 >= A \% 10 \ || \ A * 2 > B / 7$
- e. \_\_\_\_\_  $B - A < C - B / 2 \ \&\& \ A != C + 46$
- f. \_\_\_\_\_  $A + B != B - C \ || \ C / 3 <= 15$
- g. \_\_\_\_\_  $A \% 10 == 0 \ \&\& \ B \% 4 <= C \% 3$
- h. \_\_\_\_\_  $A * 3 / 4 >= 150 \ || \ A \% 3 != B \% 2$
- i. \_\_\_\_\_  $!(75 - 12 * 0.5 >= A - B - C)$
- j. \_\_\_\_\_  $!(A \% 10 < B \% 2 \ || \ C - 3 * (A - B) == 12)$

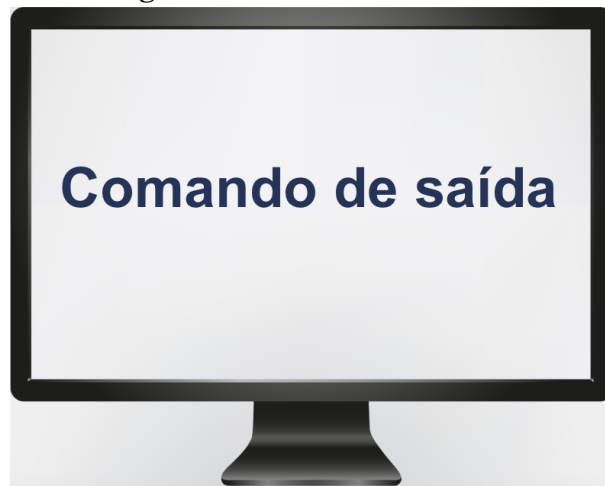
## 5 COMANDOS BÁSICOS

Este capítulo apresentará os comandos básicos para utilização de um programa. Considera-se comandos básicos aqueles que permitem uma escrita sequencial de código.

### 5.1 Comando de saída

Para que você se comunique com o usuário do programa, é importante exibir mensagens que permitam tal troca de informações. Estas mensagens são exibidas pelos comandos de saída, lembre-se de que quem está utilizando o programa não conhece o seu código.

**Figura 5 – Comando de Saída**



**Fonte: Elaborado pela autora**

Sintaxe do comando de saída

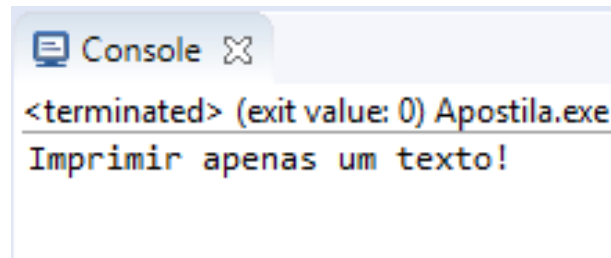
O comando responsável por exibir as informações é:

**printf** - o comando escreve o que se pede no cursor, sempre mantendo um esquema de formatação para qualquer tipo de valor.  
Ele pertence a biblioteca <stdio.h>.

A seguir exemplos para aplicar o comando de saída:

**a) Imprimir apenas texto** - Se você quiser imprimir apenas um texto fixo, basta incluir a frase, entre aspas, no comando.

```
1  #include <stdio.h>
2  int main(){
3      printf("Imprimir apenas um texto!");
4      return 0;
5  }
```

**Figura 6 – Resultado do comando de saída com texto**

Fonte: Elaborado pela autora

**b) Imprimir apenas conteúdo de variável** - Se você quiser imprimir apenas conteúdo de uma variável, basta incluir, entre aspas, a formatação referente ao tipo da variável que deseja, e, fora das aspas, indique a variável (nome) que terá seu conteúdo impresso.

```

1  #include <stdio.h>
2  int main(){
3      int idade = 12;
4      printf("%d", idade);
5      return 0;
6  }
```

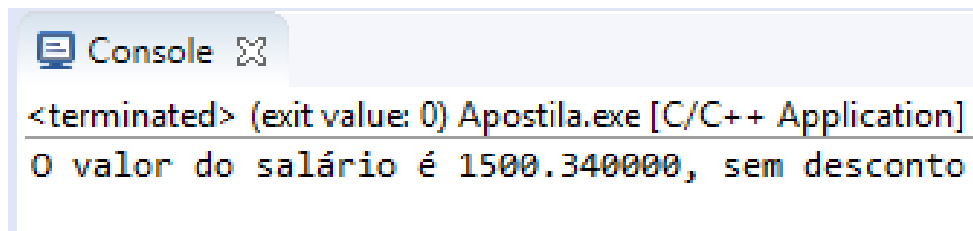
**Figura 7 – Resultado do comando de saída com conteúdo de variável**

Fonte: Elaborado pela autora

**c) Imprimir texto e conteúdo de variável** - Se você quiser imprimir um texto fixo e o conteúdo de uma variável, basta incluir ambos no comando. Você deverá criar a frase exatamente como ela ficará, e no lugar onde aparecerá o conteúdo da variável, você deve indicar o caracter de formatação do tipo desta variável.

```

1  #include <stdio.h>
2  int main(){
3      double salario = 1500.34;
4      printf("O valor do salário é %lf, sem desconto.", salario);
5      return 0;
6  }
```

**Figura 8 – Resultado do comando de saída com com texto e conteúdo de variável**


```
<terminated> (exit value: 0) Apostila.exe [C/C++ Application]
O valor do salário é 1500.340000, sem desconto
```

**Fonte: Elaborado pela autora**

A seguir um quadro com as strings de formatação (controle) de cada tipo:

**Quadro 9 – String de controle dos tipos de variáveis**

Tipo da variável	String de Controle
char	%c
cadeia de caracteres (string)	%s
int	%i ou %d
float	%f
double	%lf
endereço de memória	%p

**Fonte: Elaborado pela autora**

Você pode utilizar caracteres especiais dentro do texto, para complementar a apresentação do seu texto.

**Quadro 10 – Caracteres especiais para impressão**

Caractere especial	Função
\n	Nova linha
\t	Tabulação
\'	Imprimir aspas simples
\"	Imprimir aspas duplas
\\	Imprimir \
\0	Nulo
%%	Imprimir %

**Fonte: Elaborado pela autora**

## 5.2 Comando de entrada

Para que o seu usuário comunique-se com o seu programa, é necessário incluir o comando de entrada, ele permite que algo digitado, ou "enviado" ao computador, seja lido e utilizado no código.

**Figura 9 – Comando de Entrada**



**Fonte:** Imagem retirada da internet

O comando utilizado para receber informações do usuário é o *scanf*. Ele pertence a biblioteca *<stdio.h>*.

Sintaxe:

```
scanf("conversão", &variável);
```

onde:

**conversão** - que corresponde a *string* de controle, sempre atendendo ao tipo da variável que receberá o valor digitado.

**variável** - que corresponde ao nome da variável que receberá o valor digitado, **SEMPRE** precedido de **&**.

Para conhecermos a sintaxe do comando de entrada é necessário saber que cada tipo de variável terá uma *string* de controle específica, conforme o Quadro do comando de saída, explicado na seção anterior.

Veja abaixo os exemplos da utilização de um comando de entrada.

No exemplo só colocarei os comandos de entrada, mas lembre-se que é comum que o comando de entrada venha precedido de um comando de saída.

```
1  #include <stdio.h>
2  int main(){
3      int x; //Inclusão de um valor inteiro
4      scanf("%d", x);
5
6      float y; //Inclusão de um valor float
7      scanf("%f", y);
8      return 0;
9  }
```

## EXERCÍCIOS

Agora que já conhecemos os primeiros comandos, vamos iniciar a escrita de alguns programas. Para os enunciados que pedem códigos, não esqueça de escrevê-los em linguagem C, considerando as regras aprendidas:

1. Faça um programa (FUP) que exiba na tela o seu nome (texto fixo).

2. FUP que

Declare uma variável do tipo int

Atribua a ela o valor 45

E exiba o valor desta variável, juntamente com seu antecessor e sucessor.

3. Faça um programa que receba dois números inteiros, calcule o exiba o resto da divisão do primeiro pelo segundo.

4. Faça um programa que receba dois números reais, calcule o exiba a subtração do primeiro pelo segundo.

5. A padaria Kipão vende certa quantidade de pães franceses e uma quantidade de broas a cada dia. Cada pãozinho custa R\$ 0,62 e cada broa custa R\$ 2,90. Ao final do dia, o dono quer saber quanto arrecadou com a venda dos pães e broas (separados e juntos), e quanto deve guardar numa conta de poupança (30% do total arrecadado).

Você foi contratado para fazer os cálculos para o dono. Com base nestes fatos, faça um programa que receba a quantidade total de cada item e calcule os dados solicitados (total arrecadado por produto, total geral e valor da poupança).

6. FUP que receba o salário de um funcionário, calcule e exiba o novo salário, sabendo que houve um aumento de 25%.

7. Faça um programa que calcule e exiba a área de um quadrado Sabe-se que  $A = \text{Lado} * \text{Lado}$



## 6 CONTROLADORES DE FLUXO

A partir de agora estudaremos os comandos que permitem controlar fluxos. Quase todos já foram vistos em outras linguagens, apenas com pequenas diferenças, que serão conhecidas agora.

### 6.1 Estruturas de Decisão

Um comando de decisão permite que um código específico seja executado, considerando a condição imposta para esta execução.

#### 6.1.1 Comando *Se - if*

O comando mais utilizado para a estrutura de decisão é o *if*, ele efetua um desvio no código, considerando a condição imposta para este desvio.

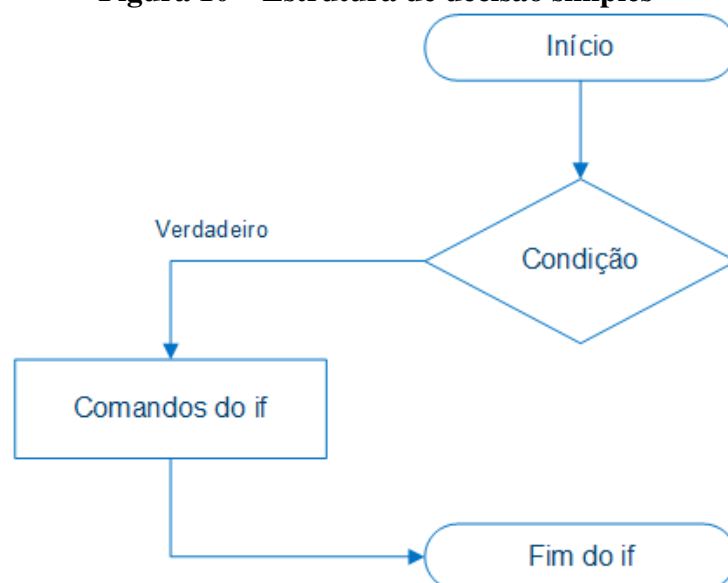
Este comando pode ser utilizado com condições compostas, simples ou aninhadas.

Vamos entender e exemplificar cada uma.

##### 6.1.1.1 Estrutura de decisão simples

O fluxograma a seguir demonstra a forma de execução de um comando de decisão simples.

**Figura 10 – Estrutura de decisão simples**



**Fonte:** Elaborado pela autora

Observe que este comando só atenderá uma situação **verdadeira**, ignorando a situação **falsa**. Na forma simples os comandos só serão executados se a condição imposta for verdadeira, não existem comandos a serem executados quando a condição for falsa.

A forma simples do comando *if* será utilizada aplicando a seguinte sintaxe:

```
if (condição) {  
    comandos executados, se a condição aplicada ao if for verdadeira.  
}
```

A condição **SEMPRE** deverá ter seu resultado como verdadeiro (*true*) ou falso (*false*). Para escrevê-la pode-se utilizar operadores relacionais e/ou lógicos, sendo que as expressões podem conter operadores aritméticos.

Quaisquer comandos poderão ficar subordinados ao *if*, mas eles só serão executados se a condição imposta for verdadeira. Por exemplo, você poderá incluir atribuições de variáveis, comandos de entrada, saída, estrutura de decisão, repetição (que ainda aprenderemos), etc.

Veja a seguir um exemplo de aplicação da estrutura de decisão simples:

```
1  #include <stdio.h>  
2  int main(){  
3      int numero;  
4      printf("Digite um número: ");  
5      scanf("%d", &numero);  
6      if (numero % 2 == 0) {  
7          printf("Número par. ");  
8      }  
9      return 0;  
10 }
```

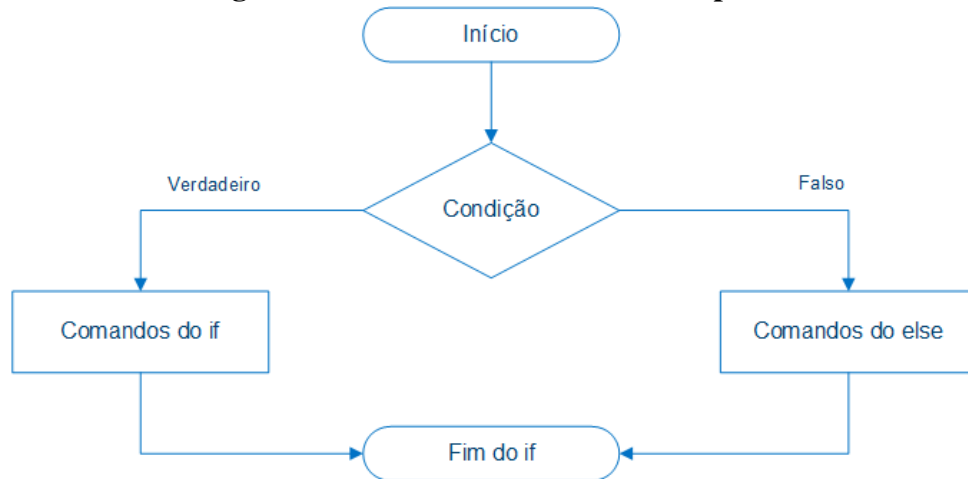
Vamos analisar o código anterior:

- > Ele irá iniciar a execução pela linha 2 (função principal);
- > Declarará a variável *numero*, reservando um espaço de memória do tipo inteiro;
- > Imprimirá na tela a mensagem "Digite um número: ";
- > Receberá, via teclado, a digitação de um valor numérico;
- > Avaliará a expressão da linha 6, verificando se ela é verdadeira ou falsa.
- > A linha 7 só será executada, se a avaliação da linha 6 for verdadeira, no caso, se o número digitado for par (esta é a avaliação).
- > A linha 9 finaliza a função principal, retornando 0 ao pré-processador, dizendo-o que o código foi executado com sucesso.

### 6.1.1.2 Estrutura de decisão composta

O fluxograma a seguir demonstra a forma de execução de um comando de decisão composto.

**Figura 11 – Estrutura de decisão composta**



**Fonte: Elaborado pela autora**

Observe que esta é a forma completa do comando, ele está preparado para atender essas duas respostas, verdadeiro (*if*) ou falso (*else*).

A forma composta do comando *if* será aplicada considerando a seguinte sintaxe:

```

if (condição) {
    comandos executados, se a condição aplicada ao if for verdadeira.
} else {
    comandos executados, se a condição aplicada ao if for falsa.
}
  
```

Seguindo a mesma lógica anterior (da forma simples), a condição **SEMPRE** deverá ter seu resultado como *true* ou *false*.

Este comando será executado sob duas situações distintas:

- a) A primeira situação ocorrerá quando a condição for **verdadeira**, e executará os comandos descritos sob o *if*.
- b) A segunda situação ocorrerá quando a condição for **falsa**, e executará os comandos descritos sob o *else*.

Veja a seguir um exemplo de aplicação da estrutura de decisão composta:

```

1  #include <stdio.h>
2  int main(){
3      int numero;
4      printf("Digite um número: ");
5      scanf("%d", &numero);
6      if (numero % 2 == 0) {
7          printf("Número par. ");
8      } else {
9          printf("Número ímpar. ");
10     }
11     return 0;
12 }

```

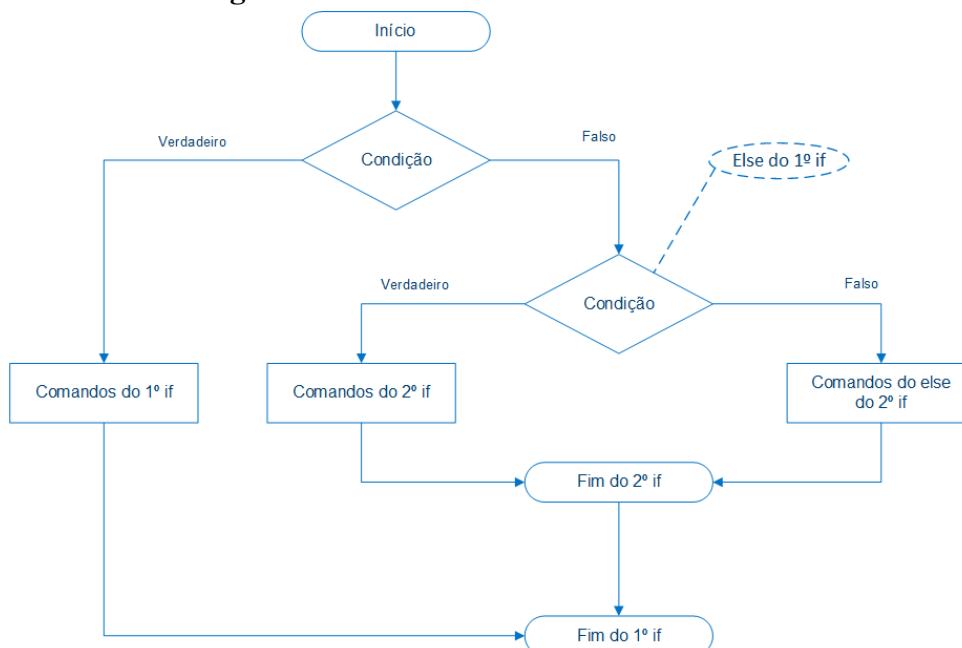
Vamos analisar o código anterior, iniciando da linha 8, pois as anteriores já foram analisadas:

-> A linha 8 possui uma indicação de senão, ou seja, se a avaliação da linha 6 for falsa, os comandos a serem executados iniciarão a partir do *else*, neste caso, somente a linha 9 será executada se a condição for falsa.

### 6.1.1.3 Estrutura de decisão aninhada

O fluxograma a seguir demonstra a forma de execução de um comando de decisão aninhada, ou seja, uma estrutura de decisão dentro de outra.

**Figura 12 – Estrutura de decisão aninhada**



**Fonte: Elaborado pela autora**

Sabendo que o comando *if* completo atende duas situações (verdadeiro ou falso), para atendermos mais condições será necessário abrir novos comandos.

Sempre que você tiver mais condições para atender deverá abrir um novo *if*. Observe que aplicar *else* faz com que o comando seja mais bem aproveitado. uma vez que já sabemos que se o comando não entrar no *if* testará o *else*, **nunca executando ambos**.

A terceira forma do comando *if* será aplicada considerando a seguinte sintaxe:

```
if (condição) {
    comandos executados, se a condição aplicada ao primeiro if for verdadeira.
} else if (condição) {
    comandos executados, se a condição aplicada ao primeiro if for falsa,
    porém, uma nova análise de condição será aplicada, por isso, esses
    comando só serão executados se a condição do segundo if for verdadeira.
} else {
    comandos executados, se a condição do segundo if for falsa.
}
```

Veja a seguir um exemplo de aplicação da estrutura de decisão aninhada:

```
1  #include <stdio.h>
2  int main(){
3      int numero;
4      printf("Digite um número: ");
5      scanf("%d", &numero);
6      if (numero < 0 ) {
7          printf("Número negativo. ");
8      } else if (numero > 0){
9          printf("Número positivo. ");
10     } else {
11         printf("Número nulo. ");
12     }
13     return 0;
14 }
```

Vamos analisar o código anterior, iniciando da linha 6, pois as anteriores já foram analisadas:

-> A linha 6 possui a avaliação, verificando se o número digitado é menor que zero.

-> A linha 7 será executada se esta avaliação for verdadeira, ou seja, se o número for menor que zero, exibindo a mensagem "Número negativo."

-> A linha 8 só será avaliada se a condição da linha 6 for falsa. Observe que nesta linha existe um comando aninhado, se a condição da linha 6 for falsa, uma nova avaliação será iniciada na linha 8, neste caso, será avaliado se o número é maior que zero.

-> A linha 6 possui a avaliação, verificando se ao número digitado é menor que zero.

Seguindo a mesma lógica anterior, a condição **SEMPRE** deverá ter seu resultado como *true* ou *false*.

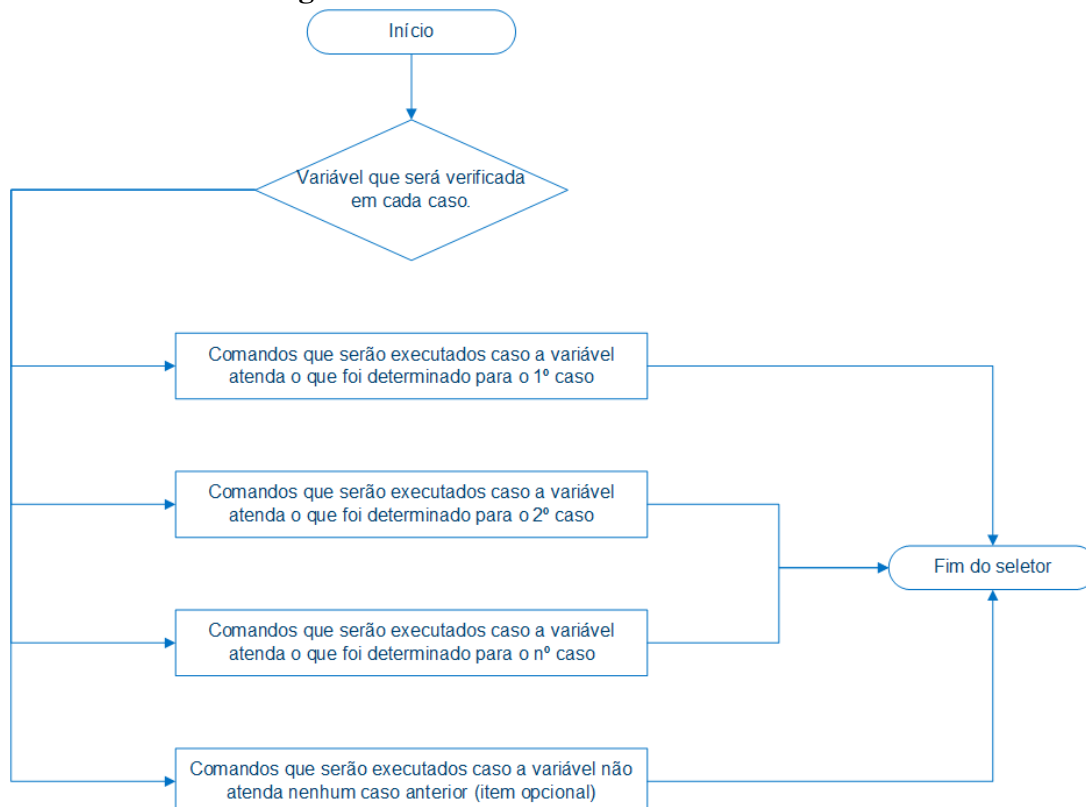
Você poderá utilizar quantos *if*'s precisar, bastando apenas abrir um novo comando. Mas lembre-se, o *else* não tem condição, quem condiciona é sempre o *if*. Esse tipo de comando será utilizado sempre que tivermos mais de duas opções para testar.

#### 6.1.1.4 Switch

O fluxograma a seguir demonstra a forma de execução de um comando de decisão que efetua seleções distintas para sua execução.

É sempre importante observar qual a sintaxe, e as exigências da linguagem que você está trabalhando. Além disto, também temos que saber se existe este tipo de comando na linguagem.

**Figura 13 – Estrutura de decisão - Seletor**



**Fonte: Elaborado pela autora**

O seletor trabalha de forma parecida com o comando *if*, ou seja, ele decidirá, a partir de uma condição, qual, ou quais, comandos serão executados, porém, ele possui limitações quanto a sua execução. Em geral, ele só atenderá valores fixos, sem possibilidade de expressões. Além disto, para a avaliação, a variável avaliada deverá ser inteira (qualquer tipo) ou caractere (char):

A sintaxe deste comando é:

```
switch (variável) {  
  
    case <constante1> :  
        <comandos1>;  
        break;  
  
    case <constante2> :  
        <comandos2>;  
        break;  
  
    case <constante3> :  
        <comandos3>;  
        break;  
  
    ...  
  
    ...  
  
    case <constanteN> :  
        <comandosN>;  
        break;  
  
    default :  
        <comandosDefault>;  
        break;  
  
}
```

Algumas observações são importantes:

- 1º. Após indicar os comandos, de cada caso, é imprescindível colocar o comando *break*, para que não teste todos os casos previstos posteriormente;
- 2º. A opção *default* é opcional, e só servirá para as situações onde nenhum caso indicado anteriormente corresponder ao valor da variável testada.
- 3º. A variável deverá atender os tipos citados anteriormente (inteira ou char), e seu valor será comparado a cada caso indicado.
- 4º. Os comandos só serão executados se atender ao caso, ou se caírem na opção *default*.

**ATENÇÃO:** Ao utilizar um comando de decisão, seja ele o *if* ou *switch*, é importante se lembrar que uma vez avaliada a condição os comandos só serão executados uma vez, ou seja, decidiu, passa para o próximo comando do código. Esta **NÃO** é uma estrutura de repetição, **APENAS** de decisão.

#### 6.1.1.5 Operador ternário - ?

O operador ternário é utilizado para simplificar a aplicação de um operador de decisão composto. Assim como este, ele está preparado para duas soluções possíveis, *true* ou *false*.

Este comando pode ser aplicado com duas sintaxes distintas:

variável = condição ? verdadeiro : falso;

OU

condição ? verdadeiro : falso;

Veja a seguir uma aplicação do operador ternário para ambas as situações:

```

1  #include <stdio.h>
2  int main(){
3      //Atribuir valor a uma variável
4      int x = 10, y = 23, z;
5      z = x > y ? x : y;
6
7      //Exibir uma mensagem
8      int numero;
9      printf("Digite um número: ");
10     scanf("%d", &numero);
11     numero % 2 == 0 ? printf("Número par. ") : printf("Número ímpar. ");
12     return 0;
13 }
```



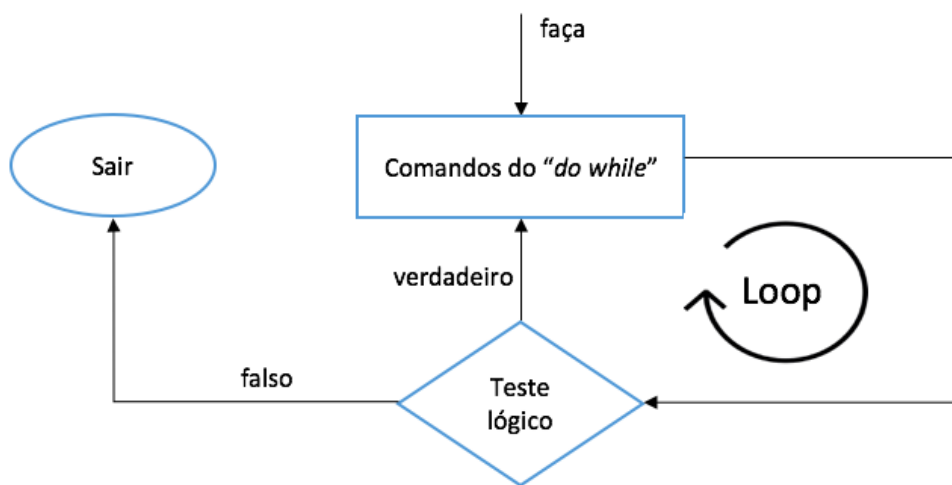
## 6.2 Estruturas de Repetição

Um comando de repetição permite que um código específico seja executado diversas vezes, dependendo da condição imposta.

### 6.2.1 Comando com condição no final - *do while*

Veja abaixo o fluxograma que representa a estrutura de repetição *do while*, que possui condicionamento no final do comando.

**Figura 14 – Fluxograma da estrutura de repetição - *do while***



**Fonte: Elaborado pela autora**

o comando *do while* efetua a execução do código e só depois testa a condição imposta, se esta condição for verdadeira, repete os comandos, caso contrário, sai da estrutura e continua no próximo comando após o final da estrutura.

A estrutura de repetição "*do while*" permite que um bloco de linhas de comandos seja repetido diversas vezes. A condição de execução é testada ao final do loop, ou seja, ao final da execução dos comandos pertencentes à estrutura.

Sua sintaxe é :

```

do {
    comandos executados, enquanto a condição aplicada ao do while for
    verdadeira.
} while (condição);
  
```

Todos os comandos serão executados uma vez, só depois da primeira execução eles serão testados, a partir daí, os comandos serão repetidos enquanto a condição for **verdadeira**.

Por isso dizemos que esta estrutura executa seus comandos, uma ou mais vezes.

Veja a seguir uma aplicação do comando *do while*, utilizando o comando para validação e para contar iterações:

```

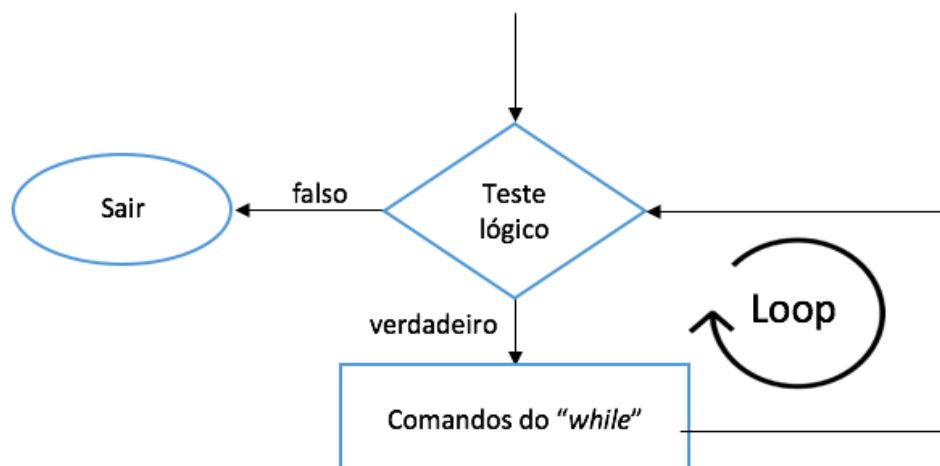
1  #include <stdio.h>
2  int main(){
3      int numero, i = 0;
4      do{
5          do{
6              printf("Digite um número (positivo): ");
7              fflush(stdin);
8              scanf("%d", &numero);
9          } while(numero <= 0);
10         if (numero % 2 == 0 ) {
11             printf("Número par. ");
12         } else {
13             printf("Número ímpar. ");
14         }
15         i++;
16     } while (i < 5);
17     return 0;
18 }

```

### 6.2.2 Comando com condição no início - *while*

Veja abaixo o fluxograma que representa a estrutura de repetição *while*, que possui condicionamento no início do comando.

**Figura 15 – Fluxograma da estrutura de repetição - *while***



**Fonte:** Elaborado pela autora

o comando *while* testa a condição imposta e, só depois, efetua a execução do código. Se esta condição for verdadeira, repete os comandos, caso contrário, sai da estrutura e continua no próximo comando após o final da estrutura.

A estrutura de repetição "*while*" permite que um bloco de linhas de comandos seja repetido diversas vezes. A condição de execução é testada no início do loop, ou seja, antes da execução dos comandos pertencentes à estrutura.

Sua sintaxe é :

```
while (condição) {
    comandos executados, enquanto a condição aplicada ao while for verdadeira.
}
```

Observe que a condição é testada antes da execução dos comandos, e estes serão repetidos enquanto a condição for **verdadeira**. Por isso dizemos que esta estrutura executa seus comandos, zero ou mais vezes.

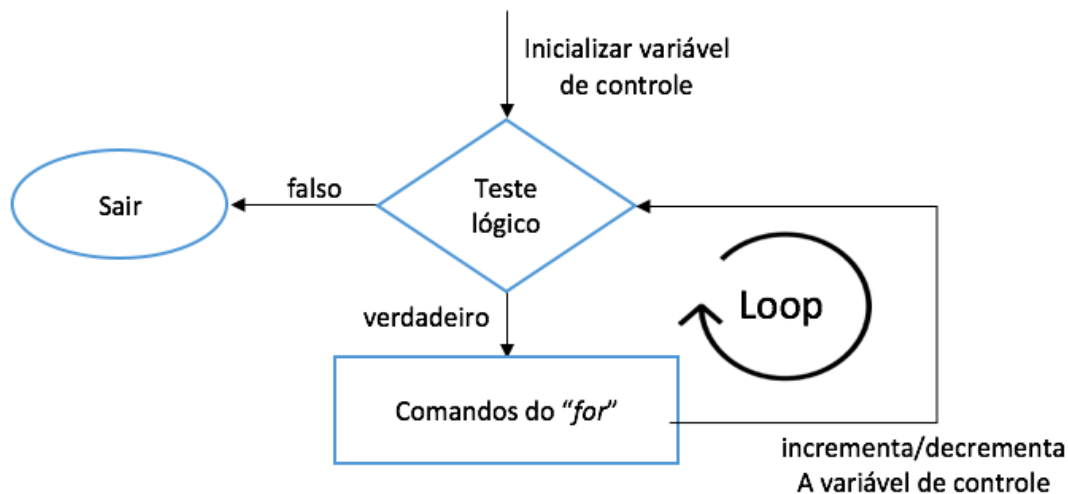
Veja a seguir uma aplicação do comando *while*, utilizando o comando para validação e para contar iterações:

```
1  #include <stdio.h>
2  int main(){
3      int numero, i = 0;
4      while (i < 5){
5          printf("Digite um número (positivo): ");
6          fflush(stdin);
7          scanf("%d", &numero);
8          while(numero <= 0); {
9              printf("Digite um número (positivo): ");
10             fflush(stdin);
11             scanf("%d", &numero);
12         }
13         if (numero % 2 == 0 ) {
14             printf("Número par. ");
15         } else {
16             printf("Número ímpar. ");
17         }
18         i++;
19     }
20     return 0;
21 }
```

### 6.2.3 Comando com contador de iterações - *for*

Veja abaixo o fluxograma que representa a estrutura de repetição *while*, que possui condicionamento no início do comando.

**Figura 16 – Fluxograma da estrutura de repetição - *while***



**Fonte:** Elaborado pela autora

o comando *for* inicializa a variável que efetua o controle de iterações, testa a condição imposta e, só depois, se esta condição for verdadeira, efetua a execução do código. Se esta condição for verdadeira, repete os comandos, caso contrário, sai da estrutura e continua no próximo comando após o final da estrutura.

A estrutura de repetição "*for*" permite que um bloco de linhas de comandos seja repetido diversas vezes.

Sua sintaxe é :

```

for (inicialização; condição; incremento/decremento) {
    comandos executados, enquanto a condição aplicada ao for for verdadeira.
}
  
```

Observe que a variável de controle é inicializada, a condição é testada antes da execução dos comandos, e estes serão repetidos enquanto a condição for **verdadeira**, a cada iteração o incremento/decremento é executado, até que a condição seja falsa. Por isso dizemos que esta estrutura executa seus comandos, zero ou mais vezes.

Veja a seguir uma aplicação do comando *for*, utilizando o comando para contar 5 iterações:

```

1  #include <stdio.h>
2  int main(){
3      int numero, i;
4      for (i = 0; i < 5; i++){
5          do{
6              printf("Digite um número (positivo): ");
7              fflush(stdin);
8              scanf("%d", &numero);
9          }while(numero <= 0);
10         if (numero % 2 == 0 ) {
11             printf("Número par. ");
12         } else {
13             printf("Número ímpar. ");
14         }
15     }
16     return 0;
17 }

```

## 6.3 Comandos auxiliares

### 6.3.1 *break*

O comando *break* é utilizado para finalizar uma estrutura de repetição, sem que a sua condição seja analisada ou para finalizar uma condição do comando *switch*.

```

1  #include <stdio.h>
2  int main(){
3      int numero, i;
4      for (i = 0; i < 5; i++){
5          printf("Digite um número (positivo): ");
6          scanf("%d", &numero);
7          if (numero < 0 ) {
8              break;
9          }
10         if (numero % 2 == 0 ) {
11             printf("Número par.");
12         }
13     }
14     return 0;
15 }

```

Este é um comando muito utilizado com *loop*'s escritos de forma eterna (que não possuem fim), mas lembre-se de que não existe *true* ou *false* nesta linguagem, estes são representados por 1 ou 0, respectivamente.

Veja abaixo dois exemplos para serem aplicados com o comando *for*:

No exemplo a seguir, o comando *for* é utilizado sem condição de parada, observe que na linha 4 do código não existe condição determinada, portanto, a única forma de finalizar a repetição é utilizando o comando *break*.

Neste caso, apesar de não finalizar o comando, é possível saber quantas iterações foram executadas, afinal, existe uma variável que efetua inicialização e contagem de execuções.

```
1  #include <stdio.h>
2  int main(){
3      int numero, i;
4
5      for (i = 0; ; i++){
6
7          printf("Digite um número (positivo): ");
8          scanf("%d", &numero);
9
10         if (numero < 0 ) {
11             break;
12         }
13
14         if (numero % 2 == 0) {
15             printf ("Número par.");
16         } else {
17             printf ("Número ímpar.");
18         }
19     }
20
21     return 0;
22 }
```

No exemplo a seguir, o comando *for* é utilizado sem inicialização, sem condição de parada e sem incremento/decremento, observe que na linha 4 do código não existe nada determinado, portanto, a única forma de finalizar a repetição é utilizando o comando *break*, e, através do comando *for*, também não será possível saber quantas iterações foram executadas.

```

1  #include <stdio.h>
2  int main(){
3      int numero, i;
4      for (i = 0; ; i++){
5          printf("Digite um número (positivo): ");
6          scanf("%d", &numero);
7
8          if (numero < 0 ) {
9              break;
10         }
11
12         if (numero % 2 == 0) {
13             printf ("Número par.");
14         } else {
15             printf ("Número ímpar.");
16         }
17     }
18     return 0;
19 }

```

### 6.3.2 *continue*

O comando *continue* é utilizado para reiniciar um *loop* em execução, voltando à condição e ignorando os comandos que estão após o *continue*.

```

1  #include <stdio.h>
2  int main(){
3      int numero, i;
4      for (i = 0; i < 5 ; i++){
5          printf("Digite um número (positivo): ");
6          scanf("%d", &numero);
7          if (numero < 0 ) {
8              continue;
9          }
10         if (numero % 2 == 0) {
11             printf ("Número par.");
12         }
13     }
14     return 0;
15 }

```

## EXERCÍCIOS

EUP que exiba um menu para o cálculo de três séries matemáticas, conforme exemplo a seguir:

a)  $\frac{1+0}{1} + \frac{2+1}{4} + \frac{3+2}{9} + \frac{4+3}{16} + \dots$

b)  $\frac{1}{1} + \frac{8}{10} + \frac{27}{100} + \frac{64}{1000} + \dots$

c)  $\frac{1}{3x2} + \frac{2}{3x4} + \frac{3}{3x6} + \frac{4}{3x8} + \dots$

d) Finalizar

Após escolher a série que será calculada, o usuário irá indicar a quantidade de termos que ele deseja calcular, este número deverá ser inteiro e positivo (valide). Após esta informação, calcule a série (considerando o número de termos solicitados) e as seguintes observações:

A série “a” será desenvolvida utilizando o comando while.

A série “b” será desenvolvida utilizando o comando do while.

A série “c” será desenvolvida utilizando o comando for.

O programa ficará em loop eterno (este poderá ser escolhido por você) até que o usuário selecione a letra “d”.

Para qualquer escolha diferente das indicadas exiba uma mensagem informando que não é uma opção válida e retorne ao menu principal, solicitando nova escolha.



## 7 VARIÁVEL COMPOSTA HOMOGÊNEA

Uma variável composta corresponde à declaração de vetores e matrizes para utilização em um código.

Estes tipos de variáveis só são utilizadas quando necessitamos armazenar valores do mesmo tipo, não sendo declaradas várias variáveis para isto.

Todas as características citadas anteriormente são utilizadas para variáveis compostas, declaração de nomes, inicialização, escopo, etc.

### 7.1 Unidimensionais - Vetores

Possuem apenas uma dimensão, também conhecida como vetor.

A sintaxe para declarar um vetor é:

```
tipo nome_do_vetor[tamanho];
```

Onde:

- > O tipo segue os valores já declarados anteriormente;
- > O nome segue as regras já declaradas anteriormente;
- > O tamanho representa a quantidade de elementos que o vetor irá conter.

É importante saber que esses elementos são indexados a partir do valor 0 (zero).

A inicialização de um vetor pode ser definida ao declará-la, bastando indicar os elementos entre chaves, separados por vírgula. Veja o exemplo a seguir:

```
int vetorExemplo[9] = {0,1,2,3,4,5,6,7,8};
```

Veja um exemplo de aplicação em código, neste caso, com preenchimento por atribuição:

```
1  #include <stdio.h>
2  int main(){
3      int vet[20], i, count = 1;
4      for(i = 0; i < 20; i++){
5          vet[i] = count;
6          count++;
7      }
8      return 0;
9  }
```

## 7.2 Bidimensionais - Matrizes

Possuem duas dimensões, definindo linha e coluna para preenchimento. Também conhecido como matriz.

A sintaxe para declarar uma matriz é:

```
tipo nome_do_vetor[linha][coluna];
```

Onde:

- > O tipo e o nome seguem as definições anteriores;
- > A linha determina quantas linhas existem na matriz;
- > A coluna determina quantas colunas existem na matriz.

A inicialização de uma matriz pode ser definida ao declará-la, bastando indicar os elementos entre chaves, separados por vírgula. Veja o exemplo a seguir:

```
int matrizExemplo[2][3] = {0,1,2,3,4,5};
```

Veja um exemplo de aplicação em código, com preenchimento por atribuição:

```
1  #include <stdio.h>
2  int main(){
3      int matriz[20][10], i, j, count = 1;
4      for(i = 0; i < 20; i++){
5          for(j = 0; j < 10; j++){
6              matriz[i][j] = count;
7              count++;
8          }
9      }
10     return 0;
11 }
```

## 7.3 Não dimensionadas

Não dimensionar uma matriz significa não indicar o tamanho que ela terá, neste caso, a inicialização definirá o valor.

Veja os exemplos a seguir:

```
double valores[] = {34.6, 23, 9.7, 12}; \\O vetor possui tamanho 4.
```

```
int numeros [][][2] = { 1,2,2,4,3,6,4,8,5,10 }; \\A matriz possui tamanho 5x2
```

## 8 MANIPULAÇÃO DE STRINGS

Na linguagem C, não existe o tipo de variável string, conforme conhecemos em outras linguagens.

Mas antes de falarmos sobre as strings, primeiro falaremos sobre os caracteres comuns.

Bem, para armazenarmos um caractere na linguagem C, utilizamos o tipo char. É importante que você saiba que sempre que faz isto, ao receber o valor alfanumérico, a linguagem também armazena seu valor decimal, ou seja, o valor correspondente na Tabela ASCII.

Veja o seguinte código:

```
1  #include <stdio.h>
2  int main(){
3      char letra = 'A';
4      printf("A letra é %c e seu valor decimal é %d., letra, letra");
8      return 0;
9  }
```

Observe que no valor de retorno foi utilizada a mesma variável, sem nenhuma alteração, porém, a string de controle foi %c para imprimir caracteres, neste caso 'A', e depois %d para imprimir decimal, neste caso 65, que corresponde a esta letra na tabela ASCII.

Agora que já conhecemos a forma de utilização de uma variável caractere, vamos falar sobre strings.

O conceito de string é um conjunto de caracteres, porém, esta linguagem não possui isto como definição, por isso, para declararmos uma string, temos que, na verdade, declarar um vetor de caracteres.

```
char vetorString[10];
```

Mas veja que esta é uma declaração de vetor comum, ou seja, no caso você está declarando um vetor que armazena até 10 caracteres.

E como definir que este vetor é uma string?

Simples, ao final da frase, ou palavra, basta incluir o caractere \0. Com esta definição, basta "procurar" o caractere para saber que a string acabou.

Pode ser que a sua string tenha vários \0, mas o código sempre procurará o primeiro, para determinar o final da sua string.

E lembre-se, no caso do nosso exemplo, a string poderá aproveitar 9 espaços para caracteres, o 10º será para o caractere \0.

**Figura 17 – Armazenamento de uma string (vetor de char)**

```
char vetorString[10];
```

0	1	2	3	4	5	6	7	8	9
T	E	S	T	E	\0				

Fonte: Elaborado pela autora

## 8.1 Funções da biblioteca string.h

Considerando a definição de string na linguagem, algumas funções já estão criadas para manipulação destas variáveis. É importante saber que para utilizá-las você deverá incluir a biblioteca string.h no código.

### 8.1.1 *gets*

Para strings, é melhor utilizar o comando de saída gets ao invés do scanf.

A sintaxe deste comando é:

```
gets(<nome da variável>);
```

Veja uma aplicação deste comando:

```

1  #include <stdio.h>
2  #include <string.h>
3  int main(){
4      char str[100];
5      printf("Digite uma string: ");
6      fflush(stdin);
7      gets(str);
8      printf("\nVocê digitou %s. ", str);
9      return 0;
10 }
```

O comando gets já completa a string, inclusive incluindo \0. Por se tratar de um vetor, você pode acessar qualquer caractere dele, bastando indicar o índice.

### 8.1.2 *strlen*

O comando retorna um valor inteiro, com a quantidade de caracteres existentes na string, anteriores ao primeiro \0 encontrado.

A sintaxe deste comando é:

```
strlen(<nome da variável>);
```

Veja uma aplicação deste comando:

```
1  #include <stdio.h>
2  #include <string.h>
3  int main(){
4      char str[100];
5      printf("Digite uma string: ");
6      fflush(stdin);
7      gets(str);
8      printf("\nA frase %s possui %d caracteres. ", str, strlen(str));
9      return 0;
10 }
```

### 8.1.3 strcpy

O comando não possui retorno, ele possui dois parâmetros que efetua a cópia da variável de origem para a variável de destino.

A sintaxe deste comando é:

```
strcpy(<nome da variável de destino>, <nome da variável de origem>);
```

Mas atenção, o valor da variável de destino será completamente substituído.

Veja uma aplicação deste comando:

```
1  #include <stdio.h>
2  #include <string.h>
3  int main(){
4      char origem[100], destino[100];
5      printf("Digite o nome: ");
6      fflush(stdin);
7      gets(origem);
8
9      strcpy(destino, origem);
10     printf("\nO nome é %s. ", destino);
11     return 0;
12 }
```

#### 8.1.4 *strncpy*

É uma função análoga á função strcpy, porém, o comando permite determinar quantos caracteres serão copiados.

O comando não possui retorno, ele possui dois parâmetros que efetua a cópia da variável de origem para a variável de destino.

A sintaxe deste comando é:

```
strncpy(<nome da variável de destino>, <nome da variável de origem>, <tamanho>);
```

Mas atenção, o valor da variável de destino será completamente substituído, na quantidade de caracteres indicados.

Veja uma aplicação deste comando:

```
1  #include <stdio.h>
2  #include <string.h>
3  int main(){
4      char origem[100], destino[100];
5      printf("Digite o nome: ");
6      fflush(stdin);
7      gets(origem);
8
9      strncpy(destino, origem, 10);
10     printf("\nO nome é %s. ", destino);
11     return 0;
12 }
```

#### 8.1.5 *strcat*

O comando não possui retorno, ele possui dois parâmetros que efetua a concatenação da variável de origem para a variável de destino.

A sintaxe deste comando é:

```
strcat(<nome da variável de destino>, <nome da variável de origem>);
```

Mas atenção, o valor da variável de destino será complementado, portanto, verifique se o tamanho da variável suportará.

Veja uma aplicação deste comando:

```

1  #include <stdio.h>
2  #include <string.h>
3  int main(){
4      char origem[100], destino[100];
5      printf("Digite o nome: ");
6      fflush(stdin);
7      gets(origem);
8
9      strcat(destino, origem);
10     printf("\nO nome é %s. ", destino);
11     return 0;
12 }
```

#### 8.1.6 *strncat*

O comando é análogo ao `strcat`, porém, permite que uma quantidade determinada seja definida.

A sintaxe deste comando é:

```
strncat(<nome da variável de destino>, <nome da variável de origem>, <tamanho>);
```

Mas atenção, o valor da variável de destino será complementado, portanto, verifique se o tamanho da variável suportará a quantidade definida.

Veja uma aplicação deste comando:

```

1  #include <stdio.h>
2  #include <string.h>
3  int main(){
4      char origem[100], destino[100];
5      printf("Digite o nome: ");
6      fflush(stdin);
7      gets(origem);
8
9      strncat(destino, origem, 15);
10     printf("\nO nome é %s. ", destino);
11     return 0;
12 }
```

### 8.1.7 *strcmp*

O comando efetua uma comparação lexicográfica entre duas strings, indicando qual é alfabeticamente maior. Por exemplo, entre os nomes "Mario" e "Maria" qual seria lexicograficamente menor, neste caso seria "Maria", vem antes de "Mario".

A sintaxe deste comando é:

```
int strcmp(<primeira variável>, <segunda variável>);
```

Os retornos possíveis são:

- 1 - Se a primeira variável for menor que a segunda.
- 0 - Se as duas variáveis forem iguais
- 1 - Se a segunda variável for menor que a primeira.

Veja uma aplicação deste comando:

```
1  #include <stdio.h>
2  #include <string.h>
3  int main(){
4      char frase[20] = "Mario", frase2[20] = "Maria";
5      printf("\n%s - %s = %d\n", frase, frase2, strcmp(frase2, frase));
6
7      return 0;
8  }
```

As funções anteriores não são todas as que pertencem a biblioteca string, apenas algumas.

É sempre importante estudarmos esta biblioteca antes de criar uma função de manipulação, pode ser que já exista o que você precisa.



## EXERCÍCIOS

Escreva um programa que execute cada item abaixo:

1. Solicite a digitação de duas strings ao usuário.
2. Acrescente a segunda string na primeira, formando uma única, e separando-as por espaço.

Exemplo:

String 1 – Estrutura

String 2 – de Dados I

String 1 – Estrutura de Dados I (após executar a atividade)

3. Exiba na tela as strings, em ordem alfabética.
4. Substitua a segunda string com o valor da primeira.

Exemplo:

String 1 – Estrutura de Dados I

String 2 – de Dados I

String 2 – Estrutura de Dados I (após executar a atividade)

5. Solicite que o usuário digite um caracter, se ele existir na string 1, substitua-o por asterisco (\*).

Exemplo:

Caracter – a

String 1 – Estrutura de Dados I

String 1 – Estrutur\* de D\*dos I (após executar a atividade)

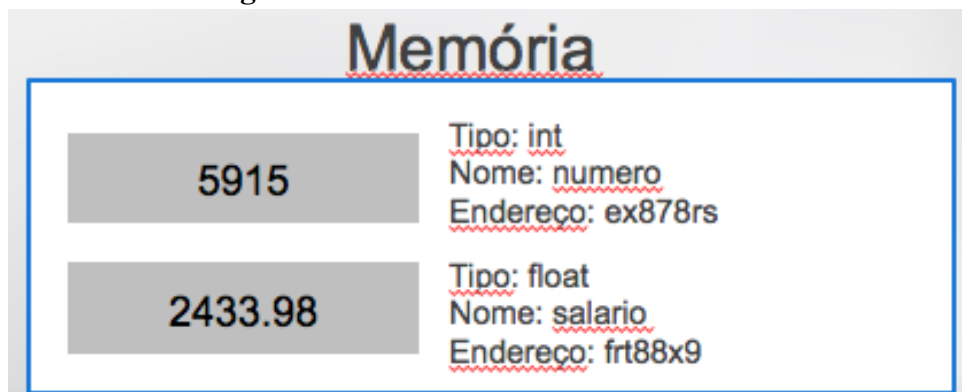
6. Exiba o tamanho da String 1.
7. Exiba o tamanho da String 2, porém, sem utilizar nenhuma função da biblioteca string.h

## 9 PONTEIROS E ALOCAÇÃO DINÂMICA

Já sabemos que uma variável corresponde a um espaço de memória, reservado para armazenar dados específicos.

Dentre as características de uma variável, existe uma, que neste caso é muito importante, O ENDEREÇO DE MEMÓRIA!! Ele permite que saibamos onde o valor está armazenado.

**Figura 18 – Armazenamento de memória**



Fonte: Elaborado pela autora

### 9.1 Conceito de ponteiro

Um ponteiro é uma variável "especial", que armazena o endereço de outra variável ou de um endereço de memória que está guardando algum dado. Isto permite que seja acessado de forma indireta ao conteúdo.

#### 9.1.1 Vantagens de utilizar um ponteiro

- \* Passagem de parâmetro por referência;
- \* Passagem de matrizes e vetores para outras funções;
- \* Manipular elementos de uma matriz ou vetor;
- \* Criar e utilizar estruturas complexas (árvores, listas, pilhas, etc);
- \* Alocar memória dinamicamente.

#### 9.1.2 Como declarar um ponteiro

A sintaxe para declaração é:

```
tipo_do_ponteiro *nome_do_ponteiro;
```

Veja um exemplo de declaração:

```

1  #include <stdio.h>
2  int main(){
3      int idade; //Declaração de uma variável comum, denominada idade
4      int *pontIdade; //Declaração de uma variável ponteiro, denominada pontIdade
5      return 0;
6  }
```

O operador `*` permite que o compilador saiba que refere-se a um ponteiro.

Para declarar um ponteiro, é importante saber que é obrigatório utilizar o mesmo tipo da variável que o ponteiro irá apontar.

Por exemplo, se você tem um variável do tipo `int`, o ponteiro que apontará para ela também deverá ser do tipo `int`.

## 9.2 Operador &

O operador `&`, quando aplicado sobre uma variável, retorna o seu endereço de memória.

Observe o exemplo abaixo:

```

1  #include <stdio.h>
2  int main(){
3      int idade, *pontIdade;
4      pontIdade = &idade;
5      return 0;
6  }
```

A linha 4 do código possui uma atribuição de valores. Neste caso, o ponteiro `pontIdade` receberá o endereço de memória da variável `idade`.

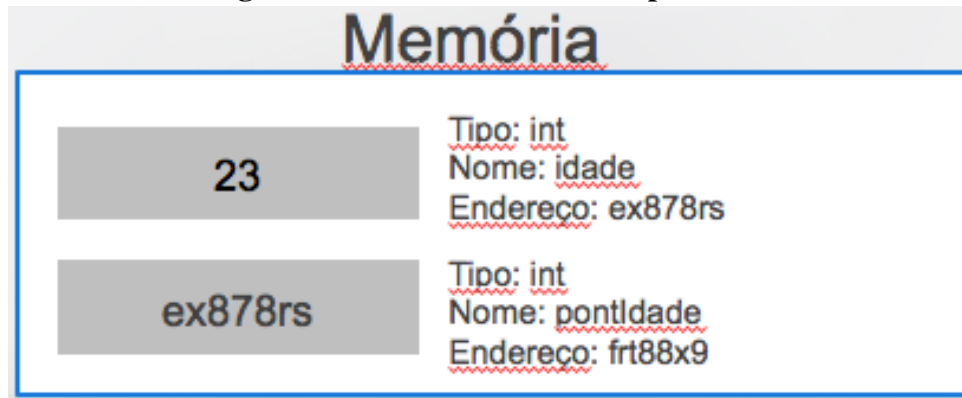
Outro exemplo muito importante é o comando `scanf`, veja o código a seguir:

```

1  #include <stdio.h>
2  int main(){
3      int idade
4      printf("Digite a idade: ");
5      scanf("%d", &idade);
6      return 0;
7  }
```

O comando *scanf*, na linha 5, exige que o operador & fique imediatamente antes da variável que receberá o valor digitado. Neste caso, o que o operador faz é passar ao comando o endereço de memória da variável para que a digitação seja armazenada no lugar correto.

**Figura 19 – Armazenamento de ponteiros**



**Fonte: Elaborado pela autora**

### 9.3 Operador \*

Este operador também é conhecido como referência de ponteiros ou operador indireto.

Pode ser utilizado de duas formas no código.

A primeira forma já foi ensinada, serve para declarar um ponteiro, indica que aquela variável não será comum, mas um ponteiro.

A segunda forma serve para manipular a variável apontada, de forma indireta, ou seja, através do ponteiro que aponta para aquela variável.

```

1  #include <stdio.h>
2  int main(){
3      int idade = 23, *pontIdade;
4      pontIdade = &idade;
5      *pontIdade = 36;
6      return 0;
7  }
```

A seguinte descrição é necessária:

- \* Na linha 3, do código acima, a variável idade recebeu o valor 23;
- \* Na linha 4, O endereço de memória da variável idade foi atribuído ao ponteiro pontIdade;
- \* Na linha 5, o valor da variável idade foi alterado de forma indireta, passou de 23 para 36, através do ponteiro.

```

1  #include <stdio.h>
2  int main(){
3      /*Declaração de uma variável comum, com o valor 23 atribuído a ela, e
4      declaração de um ponteiro. ambos do tipo int.*/
5      int idade = 23, *pontIdade;
6
7      //Atribuição do endereço de memória da variável idade para o ponteiro
8      pontIdade = &idade;
9
10     //Atribuição indireta do valor 36 à variável idade
11     *pontIdade = 36;
12
13     printf ("A variável idade possui o valor %d.", idade); //Acesso direto
14     printf ("A variável idade possui o valor %d.", *pontIdade); //Acesso indireto
15
16     printf ("Endereço de memória da variável idade %p.", &idade); //direto
17     printf ("Endereço de memória da variável idade %p.", pontIdade); //indireto
18
19     printf ("Endereço de memória da variável pontIdade %p.", &pontIdade);
20     return 0;
21 }

```

#### 9.4 Vetores x Ponteiros

Já sabemos como os vetores e matrizes são divididos na linguagem C.

Mas ao pensar em estruturas, temos que nos perguntar, "Como é dividido o endereço de memória de um vetor?".... Bem, em vetores, os endereços de memória são contíguos, ou seja, sequenciais.

Portanto, se você possui um vetor com 5 posições, de qualquer tipo, a primeira posição estará no endereço X, então, a quinta posição estará no endereço X + 4, quatro endereços após o primeiro, totalizando cinco.

**Quadro 11 – Endereçamento de vetores**

Índice	0	1	2	3	4
Conteúdo do vetor	7	19	-5	714	-20
Endereço de memória	X <sub>0</sub>	X <sub>1</sub>	X <sub>2</sub>	X <sub>3</sub>	X <sub>4</sub>

**Fonte: Elaborado pela autora**

Observação: No quadro acima, o endereço de memória é fictício, apenas para representarmos

a questão sequencial.

#### 9.4.1 Atribuição em vetores

Nas seções anteriores aprendemos a atribuir endereços de memória para um ponteiro, agora faremos isto com um vetor.

Se necessitarmos atribuir o endereço de memória de um vetor, basta atribuímos o endereço do primeiro elemento, e neste caso, o programa deve manipular o endereçamento pela quantidade de elementos de um vetor.

O endereço de memória do primeiro elemento de um vetor pode ser acessado pela seguinte sintaxe, `<nome_do_vetor>[0]`, ou apenas citando seu nome, `<nome_do_vetor>`.

Portanto, se o seu vetor tem o nome de `vetSalario`, para obter o endereço da primeira posição você pode utilizar:

`&vetSalario[0]`

OU

`vetSalario`

Veja o código a seguir:

```

1  #include <stdio.h>
2  int main(){
3      double salario[10]; //Vetor com 10 posições double, denominado salario
4      double *pontSalario; //ponteiro double, denominado pontSalario
5
6      pontSalario = &salario[0]; //Esta linha faz o mesmo que a linha 10
7
8      //OU
9
10     pontSalario = salario; //Esta linha faz o mesmo que a linha 6
11
12     return 0;
13 }
```

Obviamente, se as linhas 6 e 10 efetuam a mesma execução, não é necessário escrever ambas no mesmo código, basta escolher uma sintaxe.

### 9.4.2 Manipulação de vetores

Para manipularmos uma variável comum, podemos facilmente utilizar os operadores matemáticos. Vamos exemplificar:

Imagine que temos uma variável com o nome de idade, e com o conteúdo 23.

Se aplicarmos o seguinte comando:

```
idade++;
```

Isso significa que a variável idade será acrescida de 1 item, ou seja, de 23 passará a valer 24.

Bem, agora imagine que estamos manipulando uma variável do tipo ponteiro, que aponta para o primeiro endereço de um vetor. Vamos utilizar o código da subseção anterior para exemplificar.

A variável pontSalario aponta para o primeiro endereço do vetor salario, então, se aplicarmos o seguinte comando

```
pontSalario++;
```

A variável pontSalario será acrescida de 1 item, e como ela é um ponteiro, ela deixará de apontar para o endereço de memória atual, e apontará para o próximo sequencial.

Veja o código a seguir:

```
1  #include <stdio.h>
2  int main(){
3      int vetor[] = {34, 78, 93};
4      int *pontVetor;
5
6      pontVetor = vetor;
7      printf ("Primeiro valor - %d", *pontVetor);
8      pontVetor++;
9      printf ("Segundo valor - %d", *pontVetor);
10     (*pontVetor)++;
11     printf ("Terceiro valor - %d", *pontVetor);
12     *(pontVetor++);
13     printf ("Quarto valor - %d", *pontVetor);
14     return 0;
15 }
```

Vamos analisar o código:

**Quadro 12 – Endereçamento de vetores**

linha 3	Declaração	vetor não dimensionado
	Inicialização	34 - 78 - 93
linha 6	Atribuição	Atribui a primeira posição do vetor ao ponteiro
linha 7	Primeiro valor	Imprime o valor que está apontando pelo ponteiro - 34 (vetor[0])
linha 8	Incremento de ponteiro (Endereço de memória)	pontVetor apontava para 34 (vetor[0]) agora aponta para 78 (vetor[1])
linha 9	Segundo valor	Imprime o valor que está apontando pelo ponteiro - 78 (vetor[1])
linha 10	Incremento da variável apontada	pontVetor apontava para 78 (vetor[1]) e incrementa um nesta posição
linha 11	Terceiro valor	79, após o incremento
linha 12	Incremento de ponteiro	pontVetor apontava para 79 (vetor[1])
	O * não tem retorno	agora aponta para 93 (vetor[2])

**Fonte: Elaborado pela autora**

## EXERCÍCIOS

- O que é um ponteiro? Qual a sua utilização?
- Qual das seguintes instruções são corretas para declarar um ponteiro do tipo char?
  - char ponteiro x;
  - char \*ponteiro;
  - &char ponteiro;
  - \*ponteiro;
- Na expressão double \*ponteiro, o que é do tipo double?
  - A variável ponteiro
  - O endereço de ponteiro
  - A variável apontada por ponteiro
  - O conteúdo armazenado por ponteiro
- Assumindo que queremos ler o valor de x, e o endereço de x foi atribuído a px, a instrução seguinte é correta? Porque?
 

```
scanf("%d", *px);
```
1. Considerando que p é um ponteiro, o que aconteceria se em um código eu incluísse a seguinte linha \*(p+10);?



6. Assumindo que o endereço da variável `numero` foi atribuído a um ponteiro chamado `ponteiroNumero`. Quais das seguintes expressões são verdadeiras?

- a) `numero == &ponteiroNumero`
- b) `numero == *ponteiroNumero`
- c) `ponteiroNumero == *numero`
- d) `ponteiroNumero == &numero`

7. O programa abaixo não está funcionando, qual é a instrução que deve ser adicionada para que ele execute corretamente?

```
1    int main(){  
2        int j, *pj;  
3        *pj = 3;  
4        return 0;  
5    }
```

8. Considerando as linhas abaixo, de declaração e inicialização. Qual o valor das seguintes expressões:

```
int i = 3, j = 5;  
int *p = &i, *q = &j;
```

- a) `p == &i`
- b) `*p - *q`
- c) `**&p`

9. sabendo que `p` é um ponteiro e que aponta para uma variável comum. Explique a diferença entre as expressões a seguir:

- a) `p++;`
- b) `(*p)++;`
- c) `*(p++);`

10. Considerando as declarações, inicializações e atribuições abaixo, leia as afirmativas e marque a opção incorreta:

```
int i = 10, *pti = &i;
```

- a) `pti` armazena o endereço de `i`
- b) `*pti` é igual a 10
- c) ao se executar `*pti = 20`, `i` passará a ter o valor de 20
- d) ao se alterar de `i`, `*pti` será modificado
- e) `pti` é igual a 20

## 10 ALOCAÇÃO DINÂMICA

Você já sabe que um ponteiro permite manipulação de memória, através do armazenamento do endereço de uma variável.

Mas você já se perguntou se é possível utilizar um espaço de memória sem precisara de uma variável para "intermediar"?

Imagine que você precisa utilizar momentaneamente um espaço de memória, porém, não quer armazenar uma variável, já que aquele espaço poderá ser liberado durante a execução do programa. Isso é possível através da alocação dinâmica de memória.

### 10.1 Divisão da memória

Existem três formas de definir espaços de memória para utilização em seus códigos:

1. Variável global – O espaço existe enquanto a variável existir.
2. Variável local – Existe enquanto a função está sendo executada.
3. Alocação dinâmica – Utilizando até que o programa o libere ou termine.

Tais espaços são utilizados e divididos da seguinte forma:

Heap
Espaço de memória livre
Stack
Segmento de dados
Segmento de código

Agora vamos verificar o que cada item faz.

### ***10.1.1 Segmento de código***

Este é o espaço reservado para a execução do código do programa, todas as instruções solicitadas são efetuadas nesta área da memória.

Geralmente utilizado apenas como leitura.

### ***10.1.2 Segmento de dados***

Este é o espaço armazena variáveis globais e estáticas, seu tamanho é calculado de acordo com a necessidade das variáveis e pode ser alterado durante a execução do programa.

Seu acesso é de leitura e gravação.

### ***10.1.3 Stack***

Este é o espaço armazena as variáveis locais e as chamadas de funções, seu tamanho depende do Sistema Operacional (SO) utilizado.

Quando algum programa ultrapassa esse tamanho gera um erro de estouro de stack, mais conhecido como "*Stack buffer overflow*".

### ***10.1.4 Heap***

Este é o espaço reservado para alocação dinâmica de memória e pode ser liberada a qualquer momento.

Funções específicas efetuam o controle de todas essas ações.

## **10.2 Funções para controle da memória Heap**

Algumas funções são necessárias para controlarmos a área de memória Heap, assim, pode-se efetuar alocação dinâmica, liberando ou armazenando o conteúdo necessário.

Tais funções pertencem à biblioteca `stdlib.h`, e é necessário incluí-la no código para utilizá-las.

### ***10.2.1 Função malloc***

Este comando é responsável por alocar espaço de memória na Heap.

Sua sintaxe é:

```
void *malloc(int tamanho);
```

**void** - o tipo do comando é void, por isso, necessita de conversão (typecasting) para garantir que esteja com o tipo necessário. Após a conversão, retorna um ponteiro para a área alocada, caso contrário, retornar null.

**tamanho** - indica a quantidade de bytes que será alocada pelo comando.

```
1  #include <stdio.h>
2  int main(){
3      /*Alocação de um espaço de memória inteiro*/
4      int *x;
5      x = (int *) malloc (sizeof(int));
6
7      /*Alocação de um vetor de caracteres, 10 posições.*/
8      char *c;
9      c = (char *) malloc (10);
10
11     /*Alocação de um vetor de 5 valores double*/
12     double *y;
13     y = (double *) malloc (5 * sizeof(double));
14     return 0;
15 }
```

As linhas 5 e 13 utilizam o comando sizeof, este comando retorna a quantidade de bytes armazenada por um tipo de variável específica.

Por exemplo, se o seu SO armazena 4 bytes para uma variável do tipo inteira (int) então se utilizarmos o comando malloc ele fará exatamente este armazenamento.

### 10.2.2 Função free

Este comando é responsável por liberar o espaço de memória alocado na Heap.

Sua sintaxe é:

```
void free(void* ponteiro);
```

**void** - O comando não possui retorno

**ponteiro** - ao indicar o ponteiro ele será liberado da memória

```

1  #include <stdio.h>
2  int main(){
3      /*Alocação de um espaço de memória inteiro*/
4      int *x;
5      x = (int *) malloc (sizeof(int));
6
7      /*Alocação de um vetor de 5 valores double*/
8      double *y;
9      y = (double *) malloc (5 * sizeof(double));
10
11     /*Libera espaço de memória reservado*/
12     free(x);
13     free(y);
14
15     return 0;
16 }

```

A utilização deste comando é considerando uma boa prática de programação.

### 10.2.3 Função *calloc*

Este comando é responsável por alocar espaço de memória, em blocos, na Heap.

Sua sintaxe é:

```
void *calloc(int quantidade, int tamanho);
```

**quantidade** - quantidade de espaços reservados, considerando o tamanho indicado.

**tamanho** - indica a quantidade de bytes que será alocada pelo comando.

```

1  #include <stdio.h>
2  int main(){
3      /*Alocação de 30 espaços de memória inteiro*/
4      int *x;
5      x = (int *) calloc (30, sizeof(int));
6
7      free(x);
8      return 0;
9  }

```

### 10.2.4 Função realloc

Este comando realoca memória, anteriormente alocada pelos comandos malloc, calloc ou, até mesmo, o realloc.

Sua sintaxe é:

```
void *realloc(void *ponteiro, int tamanho);
```

**ponteiro** - Variável que está associada a alocação atual, que será realocada.

**tamanho** - Nova alteração de tamanho. Outro lugar será selecionada.

```
1  #include <stdio.h>
2  int main(){
3      /*Alocação de 30 espaços de memória inteiro*/
4      int *x;
5      x = (int *) calloc (3, sizeof(int));
6
7      /*Realocação para 30 espaços de memória inteiro*/
8      x = (int *) realloc (30, sizeof(int));
9
10     free(x);
11     return 0;
12 }
```

## EXERCÍCIOS

Faça um programa (FUP) que Solicite ao usuário o tamanho do vetor de inteiro que será criado no programa. Este tamanho deverá ser positivo (validar). Crie o vetor dinamicamente, considerando o tamanho indicado pelo usuário.

Após a criação do vetor, solicite ao usuário o preenchimento de todas as posições, com valores inteiros, positivos ou nulos. Altere as posições considerando os seguintes itens:

- \* Para as posições pares do vetor: calcule o dobro do valor digitado.
- \* Para as posições ímpares do vetor: calcule a metade do valor digitado.

Ao final:

- \* Exiba o vetor criado, com seus valores calculados e alterados
- \* Exiba a soma dos itens
- \* E não esqueça de liberar o espaço de memória utilizado.

## 11 FUNÇÕES

Até agora nosso códigos estão escritos em uma única função, porém, precisamos saber que a linguagem C permite a criação de várias funções, e principalmente, que elas trabalhem juntas.

Dividir um código em módulos, corresponde a criar várias funções que permitam um desvio no código.

Cada função será carregada uma vez e poderá ser executada a quantidade de vezes necessárias. Esta ação permite:

- \* Minimizar a quantidade de código necessário;
- \* Melhorar a estrutura;
- \* Facilitar a leitura.

### 11.1 Estrutura básica de uma função

Toda função deve contar os seguintes itens:

- \* Tipo de retorno da função
- \* Nome da função
- \* Parâmetros para receber informações externas (opcional)
- \* Comandos para a execução
- \* Retorno da função, considerando o tipo predefinido

Utilizando a seguinte sintaxe:

```
<tipo_de_retorno> <nome_da_função> (<parâmetros (opcional)>){  
    comandos de execução  
    return (que corresponde ao tipo do cabeçalho)  
}
```

Veja a aplicação em um exemplo:

```
1  #include <stdio.h>  
2  int main(){  
3      printf("Olá Mundo");  
4      return 0;  
5  }
```

Vamos avaliar cada item desta estrutura:

### 11.1.1 Tipo de retorno

O tipo de retorno de uma função indica o que ela "responderá" para quem solicitar sua execução, podendo ser qualquer tipo primitivo, ponteiro, void ou algum tipo criado pelo usuário.

Veja alguns exemplos:

**Quadro 13 – Tipos básicos para retorno de funções**

Tipo possível para retorno de função	Valor retornado no comando <i>return</i>
int	Retorna valores inteiros
float ou double	Retorna valores reais, decimais ou não.
char	Retorna caracteres (um por vez)
void	Sem retorno (não precisa do comando return)

**Fonte: Elaborado pela autora**

Uma observação importante:

Toda função dará seu retorno para quem a chamou, sendo que a função principal dará retorno para o pré-processador, finalizando o código com o valor enviado.

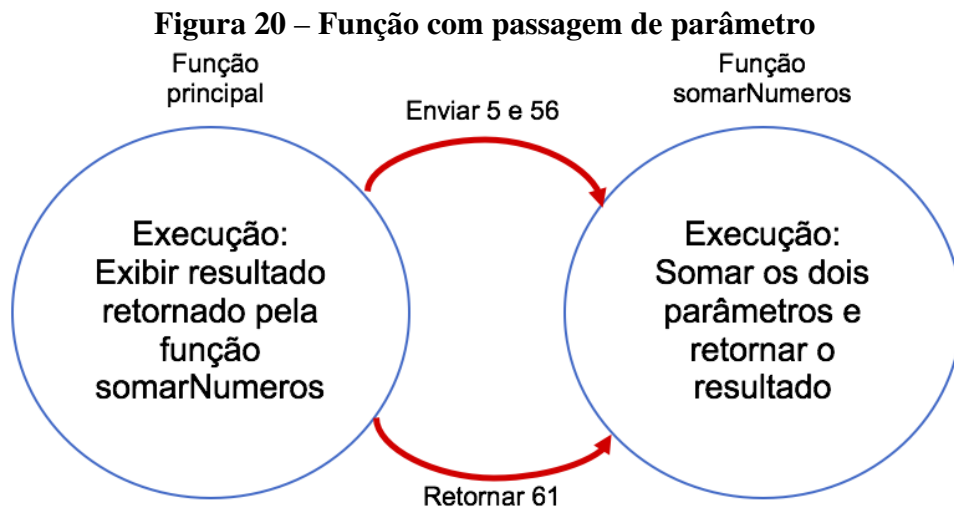
Veja um exemplo onde a função main executa uma função com retorno inteiro:

```

1  #include <stdio.h>
2
3  int somarNumeros(int numero1, int numero2);
4
5  int main(){
6      printf("%d", somarNumeros(5, 56));
7      return 0;
8  }
9
10 int somarNumeros(int numero1, int numero2){
11     return numero1 + numero2;
12 }
```



- \* O código acima inicia sua execução a partir da linha 5, na função principal.
- \* O comando printf executa a função somarNumeros, passando dois parâmetros (5 e 56), nesse momento, a função somarNumeros é executada.
- \* A função somarNumeros recebe os parâmetros enviados e retorna a soma deles, nesse momento, a função principal reinicia e o comando printf exibe o valor recebido.



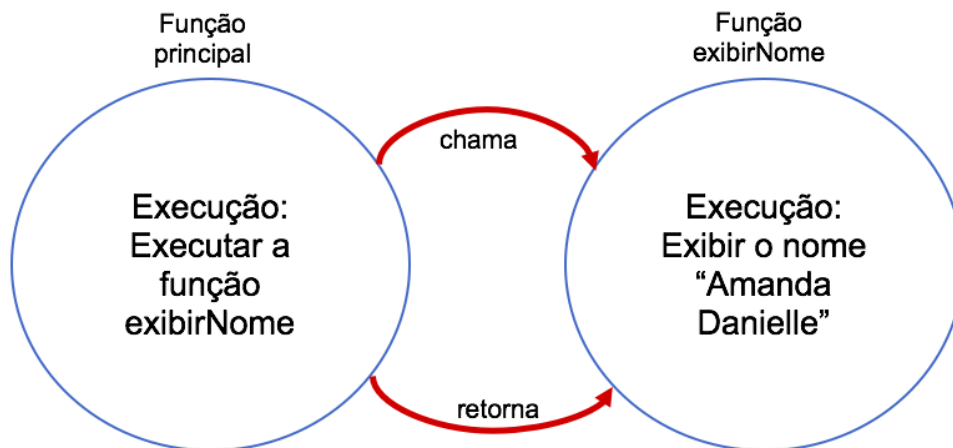
**Fonte: Elaborado pela autora**

Veja um exemplo onde a função main executa uma função sem retorno (void):

```

1  #include <stdio.h>
2
3  void exibirNome();
4
5  int main(){
6      exibirNome();
7      return 0;
8  }
9
10 void exibirNome(){
11     printf("Amanda Danielle");
12 }
```

- \* O código acima inicia sua execução a partir da linha 5, na função principal.
- \* A função exibirNome é executada, como ela não possui retorno basta apenas efetuar a chamada, nesse momento, a função exibirNome inicia-se.
- \* A função exibirNome executa um printf, exibindo um nome, e ao final, retorna para a função principal.

**Figura 21 – Função sem passagem de parâmetro**

Fonte: Elaborado pela autora

### 11.1.2 Nome

O nome da função determina como ela será reconhecida dentro do programa.

A criação deste nome segue a regra de criação de uma variável.

Para executar esta função, basta chamá-la pelo nome, incluindo seus parâmetros e respeitando seus tipos.

### 11.1.3 Protótipo da função

Toda função deve estar escrita antes da sua chamadora.

Exemplo: a função somarNumeros foi chamada dentro da função main, portanto, é imprescindível que a função somarNumeros seja criada antes da função main.

A única exceção é se você incluir o protótipo da função, ou seja, colocar a “assinatura” da função antes da função principal. Indicando ao pré-processador que ela existe e como ela funciona.

Formas de escrever um protótipo de função:

<tipo da função> <nome da função> (<parâmetros>);

OU

<tipo da função> <nome da função> (<tipos dos parâmetros>);

Veja uma aplicação destes exemplos:

Exemplo 01:

```

1  #include <stdio.h>
2  int somarNumeros(int numero1, int numero2);
3
4  int main(){
5      int resultado, numero1 = 5, numero2 = 56;
6      resultado = somarNumeros(numero1, numero2);
7      printf ("O resultado é %d", resultado);
8      printf ("Número1 = %d e Número2 = %d ", numero1, numero2);
9      return 0;
10 }
11
12 int somarNumeros(int numero1, int numero2) {
13     numero1 *= 2;
14     numero2 +=150;
15     printf ("Número1 = %d e Número2 = %d ", numero1, numero2);
16     return numero1 + numero2;
17 }

```

Exemplo 02:

```

1  #include <stdio.h>
2  int somarNumeros(int, int);
3
4  int main(){
5      int resultado, numero1 = 5, numero2 = 56;
6      resultado = somarNumeros(numero1, numero2);
7      printf ("O resultado é %d", resultado);
8      printf ("Número1 = %d e Número2 = %d ", numero1, numero2);
9      return 0;
10 }
11
12 int somarNumeros(int numero1, int numero2) {
13     numero1 *= 2;
14     numero2 +=150;
15     printf ("Número1 = %d e Número2 = %d ", numero1, numero2);
16     return numero1 + numero2;
17 }

```

A única diferença entre os exemplos 1 e 2 é a forma de criação do protótipo, porém, ambas funcionarão.

#### 11.1.4 Comandos da função

Dentro de uma função você poderá colocar qualquer outro comando, controlador ou chamada de funções existentes.

Se a função possuir retorno, não esqueça de incluir o comando `return` com o valor correspondente.

#### 11.1.5 Passagem de parâmetro

Cada função possui sua área de memória, portanto, uma não "enxerga" a outra, por isso, quando for necessário enviar informações externas para que uma função execute seus comandos, uma das opções é fazer isto via passagem de parâmetro.

Você pode passar quantos parâmetros forem necessários, e de quais tipos precisar, e inclusive, como são opcionais, se não forem necessários, não precisa passá-los.

##### 11.1.5.1 Como efetuar passagem de parâmetro

Ao efetuar a chamada da função, basta incluir o nome desta, os valores ou as variáveis, e respeitar nos respectivos locais de cada variável de parâmetro.

É importante saber que parâmetros são variáveis locais, portanto, eles terão a mesma visibilidade deles, só "valerão" dentro da função declarada, não sendo reconhecida fora dela.

Observe os códigos a seguir, eles possuem formas distintas de passagem de parâmetro.

```
1  #include <stdio.h>
2  int somarNumeros(int numero1, int numero2);
3  int main(){
4      printf ("%d", somarNumeros(5, 56));
5      return 0;
6  }
7  int somarNumeros(int numero1, int numero2){
8      return numero1 + numero2;
9  }
```

\* A função `somarNumeros` tem retorno inteiro, ou seja, espera que a resposta final seja inteira, via comando `return`.

\* Esta função também espera dois parâmetros inteiros como informação externa.

\* Na linha 7, da função `main`, existe uma chamada da função `somarNumeros`. Nesta chamada existem dois parâmetros enviados, sendo 5 e 56, respectivamente.

\* É importante respeitar a ordem de envio dos parâmetros, observe que o valor 5 preencherá a variável numero1 e o valor 56 preencherá a variável numero2.

```

1  #include <stdio.h>
2  int somarNumeros(int numero1, int numero2);
3  int main(){
4      int numero1 = 5;
5      printf ("%d", somarNumeros(numero1, 56));
6      return 0;
7  }
8  int somarNumeros(int numero1, int numero2){
9      return numero1 + numero2;
10 }
```

\* Neste exemplo, o valor da variável numero1 será enviado à variável numero1.

\* Mas atenção, a variável numero1 da função main e numero1 da função somarNumeros são distintas, cada uma está em uma área de memória diferente.

#### 11.1.5.2 Passagem de parâmetro por valor

Dizer que a passagem de parâmetro é feita por valor, é o mesmo que dizer que, apenas o valor do parâmetro será recebido dentro da função, desta forma, se você alterar qualquer valor não alterará a variável original.

Observe o código com o exemplo:

```

1  #include <stdio.h>
2  int somarNumeros(int numero1, int numero2);
3  int main(){
4      int resultado, numero1 = 5, numero2 = 56;
5      resultado = somarNumeros(numero1, numero2);
6      printf ("O resultado é %d", resultado);
7      printf ("Número1 = %d e Número2 = %d ", numero1, numero2);
8      return 0;
9  }
10 int somarNumeros(int numero1, int numero2) {
11     numero1 *= 2;
12     numero2 +=150;
13     printf("Número1 = %d e Número2 = %d", numero1, numero2);
14     return numero1 + numero2;
15 }
```

No código acima, os dois parâmetros são passados por valor, vamos analisar cada linha para compreender melhor.

**Quadro 14 – Passagem de parâmetro por valor – Análise de código**

Linha do código	Execução
4	Declaração e inicialização das variáveis resultado numero1 com o valor 5 numero2 com o valor 56
5	Chamada da função somarNumeros As variáveis numero1 e número 2 são enviadas como parâmetro A variável resultado recebe o valor retornado pela função somarNumeros
	Antes de continuar a linha 5 a função somarNumeros é executada (linha 10)
10	O parâmetro numero1 recebe o valor da variável numero1 O parâmetro numero2 recebe o valor da variável numero2 Lembre-se que são áreas de memória diferentes, portanto, variáveis diferentes
11	A variável numero1 terá o valor 10
12	A variável numero2 terá o valor 206
13	De dentro da função somarNumeros exibe a seguinte mensagem na tela: Número1 = 10 e Número2 = 206
14	Retorna a soma de numero1 e numero2 = 216 Volta para a função principal
5	A variável resultado recebe o valor 206
6	Exibe o valor 216
7	De dentro da função main exibe a seguinte mensagem na tela: Número1 = 5 e Número2 = 56

**Fonte: Elaborado pela autora**

Observe que, por serem variáveis distintas, a alteração de uma não afeta a outra. É isto que a passagem de parâmetro faz, apenas “passa” o valor para a outra área de memória, sem afetar a área principal.

Sempre que você precisar apenas informar um valor para outra função, o ideal é utilizar passagem por valor.

#### **11.1.5.3 Passagem de parâmetro por referência**

Dizer que a passagem de parâmetro é feita por referência, é o mesmo que dizer que, permitir que as variáveis utilizem o mesmo espaço de memória para efetuar as mudanças, portanto, ao alterar uma alterará a outra.

Sim, você já estudou isso antes!!! Ponteiros!!!! A passagem de parâmetro por referência é feita utilizando ponteiros, para que acessem a mesma área de memória.

Observe o código com o exemplo:

```

1  #include <stdio.h>
2  int somarNumeros(int *numero1, int numero2);
3  int main(){
4      int resultado, numero1 = 5, numero2 = 56;
5      resultado = somarNumeros(&numero1, numero2);
6      printf ("O resultado é %d", resultado);
7      printf ("Número1 = %d e Número2 = %d ", numero1, numero2);
8      return 0;
9  }
10 int somarNumeros(int *numero1, int numero2) {
11     *numero1 *= 2;
12     numero2 +=150;
13     printf("Número1 = %d e Número2 = %d", *numero1, numero2);
14     return *numero1 + numero2;
15 }

```

No código acima, o parâmetro numero1 é passado por referência e o numero2 passado por valor, vamos analisar cada linha para compreender melhor.

**Quadro 15 – Passagem de parâmetro por referência – Análise de código**

Linha do código	Execução
5	Chamada da função somarNumeros A variável numero1 é passada como parâmetro por referência, ou seja, seu endereço de memória é enviado para a outra função A variável numero2 é passada como parâmetro por valor, ou seja, apenas seu valor é passado como parâmetro A variável resultado recebe o valor retornado pela função somarNumeros
	Antes de continuar a linha 5 a função somarNumeros é executada (linha 10)
10	O parâmetro numero1 recebe o endereço de memória da variável numero1, ou seja, tudo o que acontecer com a variável da função somarNumeros, acontecerá com a variável da função principal O parâmetro numero2 recebe o valor da variável numero2
11	A variável numero1 terá o valor 10, neste caso, <b>afetará</b> a variável da função principal
12	A variável numero2 terá o valor 206, neste caso, <b>não afetará</b> a variável a função principal
13	De dentro da função somarNumeros exibe a seguinte mensagem na tela: Número1 = 10 e Número2 = 206
14	Retorna a soma de numero1 e numero2 = 216 Volta para a função principal
5	A variável resultado recebe o valor 216
6	Exibe o valor 216
7	De dentro da função main exibe a seguinte mensagem na tela: Número1 = 10 e Número2 = 56

**Fonte: Elaborado pela autora**

Observe que, o comportamento das variáveis `numero1` e `numero2` é distinto:

A variável `numero1` foi passada por referência, isto significa que sempre que você alterar o ponteiro `numero1` alterará a variável `numero1` da função principal. Sempre que precisar do valor alterado em uma função o ideal é utilizar ponteiros.

A variável `numero2` foi passada por valor, isto significa que, por serem áreas de memória distintas, alterações em uma não afetarão a outra.

## 11.2 Passagem de vetores por parâmetro

No capítulo de ponteiros, foi dito que uma das vantagens de se utilizar apontamento é poder passar vetores e matrizes por parâmetro, pois bem, agora descobriremos como.

A passagem de um vetor por parâmetro só será possível se for utilizada por referência, assim, a função que receberá a informação poderá manipular todo o vetor, desde que ela saiba o tamanho e o endereço inicial.

Veja o exemplo de um vetor de inteiros e outro do tipo `char`:

Vetor do tipo int	
Índice	Conteúdo
0	15
1	25
2	65
3	78
4	193

Vetor do tipo char	
Índice	Conteúdo
0	A
1	5
2	p

Mesmo sendo por referência, existem duas formas de utilizarmos um vetor em uma função:



### 11.2.1 Passagem de vetores por parâmetro – Aritmética de ponteiros

Em aritmética de ponteiros, a forma de manipulação dos ponteiros em um vetor ocorre através de incremento ou decremento de posições de memória. Lembre-se de que os endereços são sequenciais, portanto, é possível "caminhar" entre os endereços de memória.

É necessário saber a quantidade de posições existentes no vetor.

Veja um código de exemplo:

```

1  #include <stdio.h>
2
3  void somarElementos(int *pVet, int tam, int *soma);
4
5  int main(){
6      int soma, vet[] = { 15, 25, 65, 78 };
7      somarElementos(vet, 4, &soma);
8      printf("A soma dos elementos é %d", soma);
9      return 0;
10 }
11
12 void somarElementos(int *pVet, int tam, int *soma) {
13     int i;
14     *soma = 0;
15     for (i = 0; i < tam; i++){
16         *soma += *pVet;
17         pVet++;
18     }
19 }
```

Observe que na linha 16, o vetor recebe o incremento de uma posição de memória, assim, ele passará para a próxima posição.

Isto significa que, ao final, o ponteiro estará apontando para a posição de memória depois do vetor.

### 11.2.2 Passagem de vetores por parâmetro – Índice de vetor

Uma forma de manipular as posições de um vetor é através índice, posicionando o vetor, sempre começando do zero.

É necessário saber a quantidade de posições existentes no vetor.

Veja um código de exemplo:

```
1  #include <stdio.h>
2
3  void somarElementos(int *pVet, int tam, int *soma);
4
5  int main(){
6      int soma, vet[] = { 15, 25, 65, 78 };
7      somarElementos(vet, 4, &soma);
8      printf("A soma dos elementos é %d", soma);
9      return 0;
10 }
11
12 void somarElementos(int *pVet, int tam, int *soma) {
13     int i;
14     *soma = 0;
15     for (i = 0; i < tam; i++){
16         *soma += pVet[i];
17     }
18 }
```

Observe que na linha 16, o vetor recebe o índice de sua posição, isto significa que, ao final, o ponteiro estará apontando para a primeira posição do vetor, pois ele não será afetado, mas, ainda assim foi possível percorrer todas os índices do vetor.

## 12 DECLARAÇÃO DE NOVOS TIPOS

Além dos tipos básicos, você também pode criar seus próprios tipos, incluindo mesclando vários tipos ao mesmo tempo. Vamos ver os comandos que nos permitem fazer isto:

### 12.1 Typedef

O comando typedef permite criarmos um tipo específico, a partir de um tipo predefinido, ou seja, ele cria um tipo, que poderá ser utilizado em qualquer lugar da função, ou do programa (se o tipo for global), e que terá as características definidas pelo usuário.

Sua sintaxe é:

```
typedef antigo_nome novo_nome;
```

Primeiro exemplo:

```
1  #include <stdio.h>
2  typedef int inteiro;
3  int main(){
4      inteiro num;
5      printf("Digite um número: ");
6      scanf("%i", &num);
7      printf("O dobro de %d é %d.", num, num * 2);
8      return 0;
9  }
```

Segundo exemplo:

```
1  #include <stdio.h>
2  typedef int inteiro[4];
3  int main(){
4      inteiro num;
5      int i;
6      for (i = 0; i < 4; i++){
7          printf("Digite um número: ");
8          scanf("%i", &num[i]);
9          printf("O dobro de %d é %d.", num[i], num[i] * 2);
10     }
11     return 0;
12 }
```

Terceiro exemplo:

```

1  #include <stdio.h>
2  typedef int *inteiro;
3  int main(){
4      inteiro num;
4      inteiro num2;
4      num = &num2;
5      printf("Digite um número: ");
6      scanf("%i", &num2);
7      printf("O dobro de %d é %d.", *num, *num * 2);
8      return 0;
9  }
```

**Quadro 16 – Declaração de novos tipos**

Exemplo 1	Você utiliza a variável num, declarando-a com o tipo inteiro, porém, utiliza todas as características do tipo int.
Exemplo 2	Você utiliza a variável num, declarando-a com o tipo inteiro, porém, utiliza todas as características de um vetor int, com 4 posições.
Exemplo 3	Você utiliza a variável num, declarando-a com o tipo inteiro, porém, utiliza todas as características de um ponteiro do tipo int.

**Fonte: Elaborado pela autora**

Imagino que você está pensando o seguinte: “Mas não entendi a necessidade de criar um tipo que já existe?!” a partir de agora faremos novos tipos, mesclando os tipos já existentes, então ficará mais fácil compreender, e perceber a necessidade disto. Vamos lá?

## 12.2 Enum - Enumeration

Quando possuímos uma lista predefinida de itens, o ideal é criar um *enum* para utilizar.

Ele possui uma série de valores inteiros, que possui identificadores para representá-los.

Imagine que você precisa criar uma lista predefinida de valores inteiros, com várias "variáveis", então, a enumeração seria o ideal.

Por padrão (default), seu primeiro valor será 0 (zero), mas isto pode ser alterado, conforme sua necessidade.

Sua sintaxe é:

```
enum nome_enum {lista_de_valores};
```

Veja o exemplo abaixo:

```

1  #include <stdio.h>
2  enum diaSemana { domingo, segunda, terca, quarta, quinta, sexta, sabado };
3
4  int main(){
5      enum diaSemana d1, d2;
6      d1 = segunda;
7      d2 = quinta;
8      if (d1 == d2) {
9          printf("O dia é o mesmo.");
10     } else {
11         printf("São dias diferentes.");
12     }
13     return 0;
14 }
```

Se for necessário, você poderá definir um valor inicial para o conjunto. Veja o exemplo a seguir:

```

1  #include <stdio.h>
2  enum diaSemana { domingo = 1, segunda, terca, quarta, quinta, sexta, sabado };
3  typedef enum diaSemana diaSemana;
4
5  int main(){
6      diaSemana d1, d2;
7      d1 = segunda;
8      d2 = quinta;
9      if (d1 == d2) {
10         printf("O dia é o mesmo.");
11     } else {
12         printf("São dias d3ferentes.");
13     }
14     return 0;
15 }
16 }
```

Dessa forma, o programa considerará que os valores subsequentes serão acrescidos de 1 unidade, ou seja, a segunda terá o valor 2, a terça 3 e assim sucessivamente.

No exemplo anterior, também foi utilizado o typedef para exemplificar, assim, fica mais fácil a declaração de uma variável com as características de um enum.

### 12.3 Struct - Estruturas (Registros)

Quando necessitamos efetuar uma declaração reunindo vários tipos em uma mesma estrutura, o ideal é criar um registro.

```
struct nome_struct {
    tipo_1 nome_1;
    tipo_2 nome_2;
    . . .
    tipo_n nome_n;
};
```

Exemplo 01:

```
struct lista{
    int i;
    float f;
};
```

A estrutura anterior possui dois valores definidos, um inteiro denominado i e um float denominado f.

Exemplo 02:

```
struct tipoEndereco{
    char rua[50];
    int numero;
    char bairro[20];
    char cidade[30];
    char siglaEstado[3];
    long int cep;
};

struct fichaPessoal{
    char nome[50];
    long int telefone;
    struct tipoEndereco endereco;
};
```

Observe que uma estrutura é utilizada dentro da outra, ou seja, para cada um item da estrutura fichaPessoal é possível obter todos os campos da estrutura tipoEndereco.

Veja algumas aplicações, em todas o typedef será utilizado para facilitar a execução:

```
1  #include <stdio.h>
2  struct dados {
3      char nome[50];
4      long int telefone;
5  };
6
7  typedef struct dados dadosPessoais;
8
9  int main(){
10     //Primeira forma de utilização
11     dadosPessoais ficha01; //Declaração de estrutura simples
12     printf("Nome: ");
13     gets(ficha01.nome); //Pode ser recebido via teclado
14     ficha01.telefone = 33333333; //Pode ser atribuído
15
16     printf("Nome: %s", ficha01.nome);
17     printf("Telefone: %li", ficha01.telefone);
18
19     //Segunda forma de utilização
20     dadosPessoais ficha02[4]; //Declaração da estrutura como vetor
21     int i;
22     for (i = 0; i <= 3; i++){
23         printf("Nome: ");
24         gets(ficha02.nome[i]);
25     }
26
27     //Terceira forma de utilização
28     dadosPessoais *ficha03; //Declaração da estrutura como ponteiro
29     int i;
30     ficha03 = (dadosPessoais*) malloc (10 * sizeof(dadosPessoais));
31
32     for (i = 0; i < 10; i++){
33         printf("Nome: ");
34         gets(ficha03.nome);
35         ficha03++;
36     }
37
38     return 0;
39 }
```

## 12.4 Union - União

Em alguns casos, é necessário criar uma estrutura que reúna vários tipos mas que reaproveite a memória utilizada.

Um union faz isto, porém, você terá apenas um espaço de memória para utilizar, ou seja, só poderá um membro da estrutura por vez.

Veja o exemplo:

```
1  #include <stdio.h>
2  union valores {
3      int valI;
4      float valF;
5  };
6  typedef union valores valores;
7
8  int main(){
9      valores val;
10     val.valI = 23;
11     printf("Valor inteiro %d.", val.valI);
11
13     val.valF = 23.5;
14     printf("Valor real %d.", val.valF);
15     return 0;
16 }
```

Lembre-se, as variáveis valI e valF NUNCA serão utilizadas ao mesmo tempo, portanto, a partir da linha 13, a variável valI não estará ativa, pois a variável valF foi ativada.

## EXERCÍCIOS

### Atenção:

- Se a atividade pedir para que a função solicite, tal solicitação deve ser feita dentro do código da função.
- Se a atividade pedir para que a função receba, tal informação deve ser enviada via parâmetro.
- Não utilize variáveis globais



1. Escreva uma função (EUF) que exiba o seguinte menu:

A - Criar estrutura para preenchimento

Criar estrutura dinamicamente, considerando a quantidade de registros indicados pelo usuário na função da questão 03.

B - Preencher dados

Função da questão 4

C - Exibir produtos com estoque zerado

Preencher dados, considerando que a estrutura foi criada anteriormente e obedecendo o limite de registros informados pelo usuário.

D - Exibir produtos em estoque

Função da questão 5

E - Finalizar

Função da questão 6

Observações: O menu será exibido em loop, até que o usuário digite a opção E, para finalizar.

2. Crie uma estrutura para o cadastro de produtos com as seguintes características:

```
struct produto{
    char descricao[40];
    int codigo;
    double preco;
    int quantidade;
};
```

3. Crie uma função que solicite ao funcionário o número total de produtos (positivo) e aloque dinamicamente a quantidade de espaço necessário para efetuar o cadastro.

4. Crie uma função que efetue o cadastro de produtos com as seguintes validações.

Nome – Não nulo  
 Código – De 0 até 50  
 preço – positivo ou nulo  
 quantidade – positiva ou nula

5. Escreva uma função que exiba todos os produtos que estão faltando no estoque (quantidade zerada). Utilizando aritmética de ponteiros.
6. Escreva uma função que liste todos os produtos em estoque (quantidade não zerada). Utilizando manipulação de índice de vetor.

## REFERÊNCIAS

ASCENCIO, A. F. G.; CAMPOS, E. A. V. de. **FUNDAMENTOS DA PROGRAMAÇÃO DE COMPUTADORES**: algoritmos, pascal, c/c++ (padrão ansi) e java. 3. ed. São Paulo: Pearson Education do Brasil, 2012. 572 p.

DROZDEK, A. **ESTRUTURA DE DADOS E ALGORITMOS EM C++**. São Paulo: Thomson, 2002. 572 p.

FORBELLONE, A. L. V.; EBERSPACHER, H. F. **LÓGICA DE PROGRAMAÇÃO**: A construção de algoritmos e estruturas de dados. 3. ed. São Paulo: Pearson Prentice Hall, 2005. 213 p.

GOODRICH, M.; TAMASSIA, R. **ESTRUTURAS DE DADOS E ALGORITMOS EM JAVA**. 4. ed. Porto Alegre: Bookman, 2007. 600 p.

MIZRAHI, V. V. **ESTRUTURAS DE DADOS E ALGORITMOS**: Padrões de projetos orientados a objetos com java. São Paulo: Pearson Prentice Hall, 2008. 407 p.

MORAES, C. R. **ESTRUTURA DE DADOS E ALGORITMOS**: uma abordagem didática. 1. ed. São Paulo: Berkeley Brasil, 2001. 362 p.

PREISS, B. **TREINAMENTO EM LINGUAGEM C**. Rio de Janeiro: Campus, 2000. 566 p.

ZIVIANI, N. **PROJETO DE ALGORITMOS**: com implementações em pascal e c. 2. ed. São Paulo: Thomson, 2004. 552 p.