VICTORIO ALBANI DE CARVALHO

LÓGICA DE PROGRAMAÇÃO

COLATINA - ES
2009

Governo Federal

Ministério da Educação

Secretaria de Educação a Distância

Professor - Autor
Victorio Albani de Carvalho

Equipe Técnica
Antonio Jonas Pinotti
José Mário Costa Júnior

Revisão
Maria Isolina de Castro Soares
Antonio Jonas Pinotti

Projeto Gráfico
Moreno Cunha

Diagramação
Edson Maltez Heringer
Juliana Cristina da Silva

Crédito de Imagens (Capa e Interior)
Fonte: site sxc.hu

Ilustrador(es)
Equipe CEAD

C331*l* CARVALHO, Victorio Albani de.
Lógica de programação / Victorio Albani de Carvalho. – Colatina: CEAD / Ifes, 2009.

120p. ; il.

1. Lógica. 2. Programação. 3. Informática. 4. Educação a distância. 5. Educação profissional em nível técnico. I. Título.

CDD - 005.1

Caros alunos

Bem-vindos ao Curso Técnico de Informática que o Instituto Federal de Educação, Ciência e Tecnologia do Espírito Santo (Ifes) lhes oferece, contando com a parceria das Prefeituras e do Governo Federal.

Vocês estão de parabéns por optarem pelo ensino a distância, modalidade que está, a passos rápidos, incorporando-se à nossa cultura. O avanço acelerado das tecnologias de informação e de comunicação tem colocado as possibilidades de acesso ao conhecimento à disposição de um contingente cada vez maior de pessoas interessadas em ampliar seu campo educacional. A velocidade com que os equipamentos na área da informática têm evoluído e o aprimoramento das conexões com a web tornaram o ensino a distância uma proposta eficiente de se ensinar e de se aprender. Muitos de vocês já navegam pela rede, utilizando-se de e-mails, chats, blogs, pesquisa on-line e cursos dados por videoconferência, tornando o ambiente virtual tão familiar quanto era a sala de aula para os mais velhos.

Este curso oferecerá a vocês, alunos, material impresso e virtual de aprendizagem. Em ambos haverá teoria e variadas atividades para fixação do conhecimento.

A disciplina Lógica de Programação, oferecida no módulo I, será seu primeiro passo no aprendizado da programação de computadores. Assim, ela é de fundamental importância para o curso, pois é nela que vocês aprenderão os fundamentos básicos de programação que servirão de alicerce para todas as demais disciplinas na área de programação.

Nesta disciplina vocês aprenderão os principais conceitos de programação, como variáveis, operadores lógicos e aritméticos, estruturas de laço e estruturas de decisão e terão a oportunidade de aplicar estes conceitos na prática criando algoritmos e programas em linguagem C. Desta forma, ao final dessa disciplina vocês estarão capacitados a criar algoritmos completos e funcionais utilizando a linguagem de programação C.

Por apresentar uma metodologia flexível, nossa proposta favorece o ritmo de aprendizado de cada aluno. É preciso, no entanto, ficar atento aos prazos de sedimentação dos conteúdos e resolução das atividades avaliativas, para que vocês não se sobrecarreguem nem percam o ritmo de estudo. E lembrem-se que a melhor forma de aprender programação é praticando!

Sucesso a todos!

Victorio Albani de Carvalho e a Equipe do Ifes/e-Tec

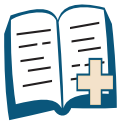
ICONOGRAFIA

Veja, abaixo, alguns símbolos utilizados neste material para guiá-lo em seus estudos.

Fala do professor.

Conceitos importantes. Fique atento!

Atividades que devem ser elaboradas por você, após a leitura dos textos.



Indicação de leituras complementares, referentes ao conteúdo estudado.

Destaque de algo importante, referente ao conteúdo apresentado. Atenção!

Reflexão/questionamento sobre algo importante, referente ao conteúdo apresentado.

Espaço reservado para as anotações que você julgar necessárias.

SUMÁRIO

Olá

Meu nome é Victorio Albani de Carvalho e sou professor especialista da disciplina Lógica de Programação neste curso a distância de Técnico em Informática.

Há 14 anos, como vocês, decidi ingressar em um curso técnico. Assim, em 1995 ingressei no curso de segundo grau técnico em Processamento de Dados na então Escola Técnica Federal do Espírito Santo, que deu origem ao atual Ifes. Foi uma decisão que mudou o rumo de minha vida pois, além de amadurecer muito como pessoa, foi meu primeiro contato com esse fascinante mundo da informática.

Depois do curso técnico nunca mais abandonei a informática, em especial, a área de engenharia de software e programação. Me graduei em Ciência da Computação e, em seguida obtive o título de Mestre em Informática pela Universidade Federal do Espírito Santo (UFES).

Atuei por 8 anos em diversas empresas exercendo várias funções, dentre elas programador, analista de sistemas e líder de projetos. Três anos atrás, ao terminar o mestrado, resolvi experimentar dar aulas e tive uma oportunidade na Faculdade Salesiana de Vitória. Lá lecionei por um ano diversas disciplinas para os cursos de Sistemas de Informação.

Em meados do ano passado fui aprovado no concurso para professor do Ifes - Campus de Colatina e hoje sou professor efetivo da coordenadoria de informática desta instituição.

Gostaria de dizer-lhes que, ao ingressar neste curso vocês estão tendo uma grande oportunidade de estudar em uma instituição de alto nível e de entrar em contato com uma das mais fascinantes áreas do conhecimento humano: a informática. Espero que vocês se apaixonem por essa área como aconteceu comigo quinze anos atrás e que se dediquem para aproveitarem ao máximo essa oportunidade.

Lembrem-se que, apesar da distância, vocês não estão sozinhos. Qualquer dúvida que tenham poderão acionar o tutor a distância e, sempre que necessário, poderão solicitar que o tutor entre em contato comigo.

O aprendizado da programação exige prática e perseverança! Assim, organizem seus tempos, dediquem-se ao curso e divirtam-se durante o processo de aprendizado.

Que tenhamos sucesso nessa jornada que se inicia!

Professor Victorio

1. INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO

Caro Aluno,

Vamos iniciar nossa viagem pelo mundo da programação. Neste primeiro capítulo, abordaremos alguns conceitos básicos e apresentaremos exemplos do nosso cotidiano a fim de facilitar o correto entendimento desses conceitos.

É importante que ao final deste capítulo você compreenda o que é um algoritmo e comece a desenvolver sua lógica de programação.

Bom estudo!

1.1. CONCEITOS BÁSICOS

Nesta disciplina, iniciaremos nossos estudos sobre Lógica de Programação. Mas, antes de começarmos, seria útil uma reflexão sobre o significado da palavra "Lógica". Assim, o que é Lógica?

A Lógica pode ser vista como a arte de pensar corretamente. A lógica visa a colocar ordem no pensamento.

Utilizamos a lógica de forma natural em nosso dia-a-dia. Por exemplo:

a) Sei que o livro está no armário.
Sei que o armário está fechado
Logo, concluo que tenho de abrir o armário para pegar o livro.

b) Sei que sou mais velho que João.
Sei que João é mais velho que José.
Então, concluo que eu sou mais velho que José.

1. Sejam os seguintes fatos:

- Todos os filhos de João são mais altos do que Maria.
- Antônio é filho de João.

Então, o que podemos concluir logicamente?

2. Considere os fatos abaixo:

- José é aluno do CEFET-ES.
- Para ser aprovado, um aluno do CEFET-ES precisa obter nota maior ou igual a 60 e comparecer a mais de 75% das aulas.
- José compareceu a todas as aulas e obteve nota igual a 80

Então, o que podemos concluir?

1.2. CONSTRUÇÃO DE ALGORITMOS

A lógica de programação é essencial para pessoas que desejam trabalhar com desenvolvimento de programas para computadores. Lógica de programação pode ser definida como um conjunto de técnicas para encadear pensamentos a fim de atingir determinado objetivo.

O objetivo fundamental de toda programação é construir algoritmos. Mas, afinal, o que é um algoritmo?

Um ***algoritmo*** é, formalmente, uma sequência finita de passos que levam à execução de uma tarefa. Podemos pensar em algoritmo como uma receita, uma sequência de instruções que dão cabo de uma meta específica.

Em outras palavras, quando criamos um algoritmo, apenas apontamos uma sequência de atividades que levam à solução de um problema. Até mesmo as soluções para os problemas cotidianos mais simples podem ser descritas por sequências lógicas de atividades, ou seja, por algoritmos:

Problema: **Trocar uma lâmpada.**

Sequência de Passos para a Solução:

1. Pegue uma escada;
2. Posicione a escada embaixo da lâmpada;
3. Pegue uma lâmpada nova;
4. Suba na escada;
5. Retire a lâmpada velha;
6. Coloque a lâmpada nova.

Esta solução é *apenas uma* das muitas soluções possíveis para o problema apresentado. Assim, ao criarmos um algoritmo, indicamos uma dentre várias possíveis sequências de passos para solucionar o problema.

Por exemplo, o problema acima poderia ser resolvido mesmo se alterássemos a sequência de passos para a sequência abaixo:

1. Pegue uma lâmpada nova;
2. Pegue uma escada;
3. Posicione a escada embaixo da lâmpada;
4. Suba na escada;
5. Retire a lâmpada velha;
6. Coloque a lâmpada nova.

3. Escreva um algoritmo (sequência de passos) para trocar um pneu de um carro.

4. Descreva um algoritmo que defina como fazer um bolo.

5. Descreva um algoritmo que defina como preparar um ovo frito.

1.3. ALGORITMOS COM ESTRUTURAS DE DECISÃO

Os algoritmos que construímos até agora apresentam uma sequência de passos que devem ser seguidos para atingir um objetivo bem definido. Note que todos os passos dos algoritmos devem ser executados a fim de que o objetivo seja alcançado.

Porém,

há algoritmos nos quais a execução de alguns passos pode depender de decisões a serem tomadas. Dessa forma, algum fato indicará se um ou mais passos do algoritmo serão executados ou não.

Por exemplo, o nosso primeiro algoritmo define uma sequência de passos para trocar uma lâmpada. Em momento algum perguntamos se a lâmpada está queimada. Simplesmente trocamos a lâmpada sem fazer qualquer teste. Para resolver esse problema, podemos acrescentar ao nosso algoritmo um teste que verifique se a lâmpada deve ser trocada:

1. Ligue o interruptor
2. Se a lâmpada não acender
   - 2.1. Pegue uma escada;
   - 2.2. Posicione a escada embaixo da lâmpada;
   - 2.3. Pegue uma lâmpada nova;
   - 2.4. Suba na escada;
   - 2.5. Retire a lâmpada velha;
   - 2.6. Coloque a lâmpada nova.

Agora, estamos ligando os passos de efetuar a troca da lâmpada a uma condição. Assim, só executaremos os passos definidos de 2.1 a 2.6 caso a condição definida do passo 2 seja verdadeira, ou seja, caso a lâmpada não acenda.

Testes que determinam quais ações serão executadas são chamados de estruturas de seleção ou estruturas de decisão.

1.3.1. Algoritmos com Estruturas de Repetição

Note que, apesar de nosso novo algoritmo estar verificando a necessidade de trocar a lâmpada antes de fazê-lo, em momento algum verificamos se a lâmpada nova que foi instalada funciona. Assim, vamos tentar alterar o nosso algoritmo a fim de garantir que ao fim de sua execução teremos uma lâmpada funcionando. Para isso, vamos incluir um novo teste em seu final:

1. Ligue o interruptor
2. Se a lâmpada não acender
   - 2.1. Pegue uma escada;
   - 2.2. Posicione a escada embaixo da lâmpada;
   - 2.3. Pegue uma lâmpada nova;
   - 2.4. Suba na escada;
   - 2.5. Retire a lâmpada velha;
   - 2.6. Coloque a lâmpada nova;
   - 2.7. Se a lâmpada não acender
     - 2.7.1. Retire a lâmpada
     - 2.7.2. Coloque uma outra lâmpada
     - 2.7.3. Se a lâmpada ainda não acender
     - 2.7.4. Retire a lâmpada
     - 2.7.5. Coloque uma outra lâmpada

     (Até quando ficaremos nesses testes???).

Pelo nosso novo algoritmo, caso a nova lâmpada não acenda, devemos trocá-la novamente e repetir esse procedimento indefinidamente até que uma lâmpada funcione. Note que não sabemos quantas vezes teremos de repetir o teste até acharmos uma lâmpada que funcione.

Em casos como esse, devemos utilizar estruturas de repetição. Essas estruturas definem um fluxo de ações que se repetem enquanto uma determinada situação acontece.

Dessa forma, substituímos nossa sequência indefinida de estruturas de decisão por uma estrutura de repetição:

1. Ligue o interruptor;
2. Se a lâmpada não acender:
   2.1. Pegue uma escada;
   2.2. Posicione a escada embaixo da lâmpada;
   2.3. Pegue uma lâmpada nova;
   2.4. Suba na escada;
   2.5. Retire a lâmpada velha;
   2.6. Coloque a lâmpada nova.
   2.7. Enquanto a lâmpada não acender:
      2.7.1. Retire a lâmpada
      2.7.2. Coloque uma outra lâmpada.

Assim, neste novo algoritmo, enquanto a condição definida na linha 2.7 for verdadeira (ou seja, enquanto a lâmpada não acender), as ações definidas em 2.7.1 e 2.7.2 serão repetidas.

6. Elabore um algoritmo que indique como fazer uma prova. Faça o algoritmo pensando que o aluno não deve deixar questões em branco, assim deve continuar fazendo a prova enquanto existir questão em branco e o tempo de prova não tiver acabado. Além disso, o aluno só deve resolver uma questão se souber resolvê-la, senão pula para a próxima.

7. Suponha que você tenha uma caixa cheia de bolas. Nessa caixa existem bolas azuis e bolas vermelhas. Além disso, você tem também duas caixas vazias. Vamos chamar a caixa que contém as bolas de "caixa 1" e as duas caixas vazias de "caixa 2" e "caixa 3". Neste contexto, escreva um algoritmo que defina como tirar todas as bolas da "caixa 1" colocando as bolas azuis na "caixa 2" e as bolas vermelhas na "caixa 3".

8. José trabalha no departamento de recursos humanos de uma empresa. A empresa de José definiu que os salários dos empregados serão aumentados seguindo a seguinte regra: caso o salário seja menor que R$1000,00 o aumento será de 10% e, caso contrário, será de 8%. José recebeu uma lista contendo os nomes e salários de todos os funcionários da empresa e foi solicitado que calculasse o novo salário desses funcionários. Assim, escreva um algoritmo para que José calcule corretamente os novos salários.

9. Desafio de lógica: Três missionários e três canibais encontram-se na margem esquerda de um rio. Nessa margem também existe um bote que pode transportar uma ou duas pessoas. As seis pessoas pretendem todas passar para a margem direita (usando o bote). No entanto, os missionários têm de arranjar um plano para o conseguir de modo que, em nenhuma circunstância, existam missionários numa margem em minoria relativamente aos canibais, pois têm receio do que lhes possa acontecer. Quando o bote chega à margem, os elementos do bote são contados como estando na margem. Assim, se houver apenas um canibal em uma margem, não podemos enviar a essa margem o bote com um canibal e um missionário, pois, ao chegar à outra margem, serão dois canibais contra um missionário. Lembre-se de que para o bote ir de uma margem a outra é necessário que alguém esteja remando, ou seja, o bote nunca atravessa vazio. Faça um algoritmo que exiba, passo a passo, como efetuar esta travessia de forma segura.

2. CONSTRUÇÃO DE ALGORITMOS PARA COMPUTADORES

Caro Aluno,

Vamos iniciar a segunda etapa da nossa viagem. Agora começaremos a aprender os conceitos básicos referentes à programação de computadores. Desenvolveremos algoritmos utilizando o Portugol: uma linguagem utilizada apenas para introduzir os conceitos de programação, por ser mais simples que as linguagens de programação mais utilizadas.

Ao final deste capítulo você será capaz de desenvolver algoritmos muito parecidos com os programas interpretados por computadores.

Vamos em frente!

2.1. FORMALIZANDO A ESCRITA DE ALGORITMOS

Para que os computadores sejam capazes de interpretar os algoritmos que desenvolvemos, precisamos transformar a sequência de passos que escrevemos em linguagem natural para uma linguagem que possa ser "entendida" pelo computador. Essas linguagens são chamadas de **linguagens de programação**. Existem diversas linguagens de programação em uso atualmente. Em nosso curso, a partir do capítulo 3, utilizaremos a Linguagem C.

Neste capítulo, a fim de facilitar o aprendizado dos principais conceitos de programação, utilizaremos o Portugol. O Portugol é uma linguagem que une o formalismo das linguagens de programação à facilidade de compreensão da linguagem natural.

Para entender a construção de algoritmos nessas linguagens, vamos iniciar estudando alguns conceitos básicos como variáveis e constantes.

2.2. VARIÁVEIS

O primeiro passo para que um programa seja executado em um computador é o carregamento desse programa para a memória. A memória é utilizada para armazenar tanto as instruções dos programas quanto os dados utilizados pelos mesmos. Qualquer programa, para ser executado, tem de estar na memória (FARRER, 1999).

Memória:

Meio físico para armazenar dados temporariamente ou permanentemente (TANENBAUM, 1997, p.212).

Ao desenvolvermos nossos algoritmos, frequentemente precisamos armazenar dados referentes ao problema, como um nome, um número ou mesmo o resultado de uma operação. Mas, para armazenar esses dados, precisamos solicitar ao computador que ele reserve uma área da memória para nosso uso. A forma de solicitar ao computador que reserve memória é chamada de **declaração de variáveis.**

A sintaxe para a declaração de variáveis em Portugol é dada abaixo:

Sintaxe: var tipo_da_variavel nome_da_variavel;

var

A palavra *var* é utilizada em Portugol para indicar que estamos declarando uma variável.

tipo_da_variavel

Precisamos informar o *tipo de dados* que armazenaremos na variável para que o computador saiba o tamanho do espaço de memória que reservará. A seguir (item 2.2.2) serão apresentados os tipos que podem ser utilizados.

nome_da_variavel

Quando solicitamos que o computador reserve espaço de memória, temos de informar como vamos nos referir a essa área de memória reservada, ou seja, *qual nome* daremos a esse espaço de memória. Assim, toda variável tem um nome através do qual é referenciada. A seguir (item 2.2.1) apresentaremos um conjunto de regras que devem ser seguidas para dar nomes às variáveis.

Todos os comandos em Portugol são finalizados com ; (ponto-e-vírgula).

2.2.1. Por que declarar variáveis e como nomeá-las?

Sempre que criamos uma variável, nós o fazemos com o objetivo de armazenar algum tipo de valor específico. Por exemplo, se estivermos desenvolvendo um algoritmo que calcule o imposto de renda a ser pago por um assalariado, precisaremos de variáveis para armazenar o valor do salário, bem como para armazenar os resultados dos cálculos. Assim, o nome dado à variável deve deixar claro o objetivo da mesma, ou seja, devemos utilizar nomes sugestivos. Apesar de esta ser a principal diretriz quanto à atribuição de nomes a variáveis, algumas outras regras são apresentadas a seguir:

REGRA	EXEMPLO
Inicie sempre por um caractere alfabético, nunca por um número.	Nome (correto) - 1nome (errado)
Não utilize caracteres especiais como “, ( ) / *; +.	Nome (M); N*B
Não coloque espaços em branco ou hífen entre nomes.	salario-bruto
Utilize, se necessário, *underline*	salario_bruto
Crie suas variáveis com nomes sugestivos.	Se vai guardar salário de funcionários, dê à variável o nome *salario*.

Tabela 1 – Como nomear variáveis

2.2.2. O que são tipos de variáveis?

Quando declaramos uma variável, devemos ter em mente os valores que serão armazenados naquele espaço de memória. É essa observação que definirá o tipo da variável a ser declarado. Uma variável pode ser de um dos seguintes tipos:

- ***Tipo inteiro***: Declararemos variáveis do tipo *numérico inteiro* quando precisarmos armazenar valores inteiros, positivos ou negativos (1, 5, 7, -10, -5, ...). Por exemplo, se precisarmos de uma variável para armazenar o número de filhos de um funcionário, o tipo ideal para essa variável seria *inteiro*.
- ***Tipo real***: Declararemos variáveis do tipo *numérico real* para armazenar valores reais, em outras palavras, valores com ponto decimal (5.7, 3.2, -8.5). Esse seria o tipo ideal para armazenar, por exemplo, o salário de funcionários.
- ***Tipo caractere***: Declararemos variáveis do tipo *literal caractere* para armazenar um único caractere, que pode ser uma letra ou um símbolo. Por exemplo, para identificar o sexo do indivíduo, armazenaremos apenas o caractere ‘F’ ou ‘M’.
- ***Tipo cadeia***: Declararemos variáveis do *tipo literal cadeia* para armazenar uma sequência de caracteres, ou seja, uma palavra, uma mensagem, um nome. Assim, se precisarmos de uma variável para armazenar o nome de uma pessoa, esse seria o tipo ideal.
- ***Tipo lógica***: Declararemos variáveis do *tipo lógico* para armazenar valores lógicos, ou seja, o valor de variáveis desse tipo será sempre VERDADEIRO ou FALSO.

Variável:

Uma Variável é uma posição nomeada de memória, que é usada para guardar um valor que pode ser modificado pelo programa. (LAUREANO, 2005, p. 12).

Lembrando a sintaxe:

var tipo_da_variavel nome_da_variavel;

Assim, a declaração de uma variável para armazenar o salário de um funcionário pode ser feita da seguinte forma em Portugol:

var real salario;

10. Aprendemos algumas regras que devem ser seguidas para dar nomes a variáveis. Assinale os nomes de variáveis que obedecem a essas regras:

a) ( ) nome
b) ( ) telefone-celular
c) ( ) nome+sobrenome
d) ( ) 2taxa
e) ( ) telefone_celular
f) ( ) conta1

11. Para cada valor dado abaixo, foi definido um tipo de variável. Marque os pares de valor e tipo definidos corretamente:

a) ( ) valor = 2.5 tipo = real
b) ( ) valor = 'F' tipo = inteiro
c) ( ) valor = -2 tipo = inteiro
d) ( ) valor = 'M' tipo = caractere
e) ( ) valor = 5 tipo = cadeia
f) ( ) valor = -10.35 tipo = real
g) ( ) valor = 38 tipo = real
h) ( ) valor = 'Jose' tipo = cadeia
i) ( ) valor = 135 tipo = inteiro
j) ( ) valor = 7.5 tipo = inteiro

12. Você está fazendo um algoritmo para calcular a média dos alunos a partir das notas de 2 provas. Assim, precisará de 3 variáveis: uma para a nota da primeira prova, uma para a nota da segunda prova e uma para a média. Segundo as normas da instituição, as notas das provas devem ser números inteiros de 0 a 10. Já para a média podem ser atribuídos valores com casas decimais. Utilizando a sintaxe de declaração de variáveis em Portugol e as regras para definição de tipos e de nomes, indique como você declararia essas 3 variáveis. ***Dica: lembre-se de escolher nomes sugestivos para as variáveis.***

2.3. CONSTANTES

Como aprendemos, o valor de uma variável pode ser alterado ao longo de seu algoritmo. Mas, às vezes, precisamos armazenar valores que não se alteram. Para isso existem as constantes.

As constantes são criadas obedecendo às mesmas regras já vistas para *variáveis.* Diferem apenas no fato de armazenar um valor constante, ou seja, que não se modifica durante a execução de um programa.

A sintaxe para a declaração de constantes em Portugol é dada abaixo:

Sintaxe: ***const nome_da_constante ← valor;***

const

A palavra **const** é utilizada em Portugol para indicar que estamos declarando uma constante.

nome_da_constante

É o nome através do qual vamos nos referir à constante. Deve obedecer às mesmas regras que os nomes de variáveis. Apenas para ficar fácil a diferenciação entre variáveis e constantes em seus programas, aconselha-se que todas as letras dos nomes das constantes sejam maiúsculas.

valor

É o valor assumido pela constante.

Como exemplo, considere um algoritmo que calcule o valor da contribuição do FGTS: 8% sobre o salário, independentemente do valor do salário. Assim, a taxa de 8% será constante durante a execução do programa. Logo, poderia declarar a constante da seguinte forma:

const TAXA_FGTS = 0.08;

Constante:
Variável com valor pré-definido que não pode ser modificado por nenhuma função de um programa. (LAUREANO, 2005, p.16).

2.4. COMANDOS DE ATRIBUIÇÃO, ENTRADA E SAÍDA DE DADOS

2.4.1. Comandos de Atribuição

Na construção de algoritmos, depois que declaramos nossas variáveis e constantes, geralmente precisamos indicar que elas armazenarão um determinado valor durante a execução do programa. Para isso, utilizamos o comando de atribuição que, em Portugol, é representado por uma seta (←), conforme sintaxe abaixo:

Sintaxe: ***identificador ← expressão;***
identificador
Nome da variável ou constante a ser utilizada
expressão
Valor ou expressão a ser armazenado (veremos ainda neste capítulo que podemos ter expressões aritméticas e expressões lógicas).

Exemplo:

```
var inteiro nota; ..... /*criamos a variável inteira nota*/
nota ← 10; ..... /*atribuímos o valor 10 à variável nota*/
var caractere sexo; ... /*criamos a variável caractere sexo*/
sexo ← 'F'; ..... atribuímos o caractere 'F' à variável sexo*/
```

2.4.2. Comando de Entrada de Dados

Frequentemente, na construção de algoritmos, precisamos solicitar que usuários informem, por meio do teclado, alguns valores a serem utilizados durante a execução. Por exemplo, se fizermos um algoritmo para calcular a média das notas de um aluno, precisaremos solicitar quais foram as notas, para depois calcularmos a média. Esses valores informados devem ser armazenados em variáveis para que sejam utilizados quando necessário.

O comando de entrada de dados será responsável pela leitura e armazenamento desses dados na variável que indicarmos. A sintaxe do comando de entrada de dados em Portugol é exibida a seguir:

Sintaxe: ***leia (variavel);***

leia ( )

Função responsável por ler o que o usuário digitou e armazenar o valor na variável indicada.

variavel

Nome da variável utilizada para armazenar o valor digitado.

Exemplo:

```
var inteiro nota; ....../*criamos a variável inteira nota*/
leia (nota); ....../*atribuímos à variável nota o
                    valor que o usuário digitar*/
```

2.4.3. Comando de Saída de Dados

Através da utilização do comando de saída de dados conseguimos exibir mensagens ou valores para o usuário de nossos programas. É através desse comando que nosso algoritmo consegue se comunicar com os usuários para solicitar a entrada de dados ou para fornecer saídas de dados.

O comando de saída de dados exibe no monitor valores de constantes, variáveis ou expressões. A sintaxe do comando de saída de dados em Portugol é exibida abaixo:

Sintaxe: ***escreva (expressao);***

escreva ( )

Função responsável por escrever no monitor uma mensagem para o usuário.

expressao

Indica o que será escrito no monitor. É normalmente composta por um texto fixo seguido por uma vírgula e um nome de variável.

Exemplo:

```
Algoritmo exemplo;
inicio
    var cadeia nome; ...../*criamos a variável nome do tipo cadeia*/
    escreva ("Digite seu Nome"); ..../*solicitamos que o usuário digite
                                       seu nome*/
    leia (nome); ...../*lemos para a variável nome o
                        valor digitado pelo usuário*/
    escreva ("Bom dia", nome); ...../*escrevemos na tela a mensagem
                                      Bom dia acompanhado pelo nome
                                      digitado pelo usuario*/
fim
```

Assim, caso o usuário tenha digitado o nome Maria, será exibida no monitor a seguinte mensagem: *Bom dia Maria*.

13. Assinale os comandos de atribuição realizados corretamente:

a) ( ) var cadeia SEXO ← 'F';
b) ( ) var inteiro ALTURA ← 1,80;
c) ( ) var real SALÁRIO ← 3.000,00;
d) ( ) var cadeia ← "NOME";

14. No programa abaixo, dois valores inteiros são lidos e somados e o resultado dessa soma é mostrado no final da execução. Analise as linhas do programa e assinale as afirmações corretas:

linha 1 ... Algoritmo soma;
linha 2 ... inicio
linha 3 ... **var** int NUM1, NUM2, SOMA;
linha 4 ... **escreva (**"Digite o primeiro número");
linha 5 ... **leia (**NUM1);
linha 6 ... **escreva** ("Digite o segundo número");
linha 7 ... **leia** (NUM2);
linha 8 ... SOMA ← NUM1 + NUM2;
linha 9 ... **escreva** ("A soma dos números digitados é:", SOMA);
linha 10 ... **fim**

a) ( ) linha 5 → O primeiro valor digitado no teclado está sendo lido e armazenado em NUM1;
b) ( ) linha 7 → O segundo valor digitado no teclado está sendo lido e armazenado em NUM2;
c) ( ) linha 8 → O resultado da soma dos valores digitados está sendo atribuído à variável SOMA;
d) ( ) linha 9 → No monitor serão exibidas a mensagem que está entre aspas e a soma dos números digitados;

15. Faça um algoritmo que solicite que o usuário digite seu nome e a seguir solicite que seja digitada sua idade. Depois que o usuário digitar o nome e a idade, o programa deve exibir na tela duas mensagens: uma com o nome e outra com a idade do usuário. Suponha que o usuário seja o Pedro e tenha 32 anos. Assim, após a digitação dos dados, seu programa deve exibir as seguintes mensagens: *"Seu nome é Pedro"* e *"Você tem 32 anos"*.

2.5. OPERADORES ARITMÉTICOS E EXPRESSÕES ARITMÉTICAS

Os operadores aritméticos são símbolos que representam operações aritméticas, ou seja, as operações matemáticas básicas. Abaixo é apresentada uma tabela contendo os operadores aritméticos que utilizaremos neste curso.

OPERADOR	OPERAÇÃO MATEMÁTICA
+	Soma
–	Subtração
*	Multiplicação
/	Divisão
%	Resto da Divisão Inteira

Tabela 2 – Operadores Aritméticos

Os operadores aritméticos serão utilizados para formar expressões aritméticas. As expressões aritméticas são formadas por operadores aritméticos que agem sobre operandos. Os operandos podem ser variáveis ou constantes do tipo numérico, ou seja, inteiros ou reais. Abaixo temos dois exemplos de expressões aritméticas:

nota/2

x*2+y/2

Na resolução de expressões aritméticas deve-se utilizar a seguinte ordem de precedência entre os operadores:

PRIORIDADE	OPERADOR	OPERAÇÃO
1ª	* / %	multiplicação, divisão, resto da divisão
2ª	+ -	adição, subtração

Tabela 3 – Ordem de Precedência entre Operadores Aritméticos

Além da ordem de prioridades definida acima, podemos utilizar parênteses. Assim, resolvemos primeiro as expressões contidas nos parênteses mais internos, seguindo a ordem de precedência entre operadores, passando depois para os parênteses mais externos. Por exemplo, na expressão:

nota1 + (nota2 + nota3) / 2

Primeiro somamos nota2 a nota3; o resultado é divido por 2 e só depois somamos com nota 1.

Um fato interessante é que o resultado da execução de expressões aritméticas é sempre um valor numérico (inteiro ou real) que pode então ser atribuído a uma variável numérica através do uso do comando de atribuição estudado anteriormente.

Resumindo:

- expressões aritméticas: uma conta a ser feita
- operadores aritméticos: os sinais + - * / %
- operandos aritméticos: constantes ou variáveis (inteiras ou reais), ou outra expressão aritmética
- precedência na ordem dos cálculos: a mesma da matemática, podendo haver vários níveis de parênteses; não há colchetes [ ] ou chaves { }.

Como exemplo, vamos criar um algoritmo para ler e multiplicar dois números inteiros e exibir o resultado.

É importante observar cada linha desta sequência:

linha 1 ... **Algoritmo** multiplicacao;
linha 2 ... **inicio**
linha 3 ... **var** inteiro NUM1, NUM2, MULT;
linha 4 ... **escreva** (“Digite o primeiro número”);
linha 5 ... **leia** (NUM1);
linha 6 ... **escreva** (“Digite o segundo número”);
linha 7 ... **leia** (NUM2);
linha 8 ... MULT ← NUM1 * NUM2;
linha 9 ... **escreva** (“O resultado da multiplicação é:”, MULT);
linha 10 ... **fim**

Vamos entender todas as linhas do nosso algoritmo:

linha 1 ... Nome do programa.

linha 2 ... Indica o início do programa.

linha 3 ... Declaração das três variáveis do tipo inteiro necessárias ao programa.

linha 4 ... O comando ***escreva*** exibirá a mensagem que solicita a digitação do primeiro número.

linha 5 ... O primeiro número digitado será lido e armazenado na variável NUM1.

linha 6 ... O comando ***escreva*** exibirá a mensagem que solicita a digitação do segundo número.

linha 7 ... O segundo número digitado será lido e armazenado na variável NUM2.

linha 8 ... A variável MULT receberá o resultado da multiplicação do primeiro pelo segundo número.

linha 9 ... O comando ***escreva*** exibirá uma mensagem com o resultado da multiplicação.

linha 10 ... Indica o fim do programa.

2.6. OPERADORES RELACIONAIS, OPERADORES LÓGICOS E EXPRESSÕES LÓGICAS

Para entender o conceito e o uso de expressões lógicas, primeiro precisamos conhecer os operadores lógicos e os operadores relacionais, pois as expressões lógicas são formadas a partir da utilização desses operadores.

2.6.1. Operadores Relacionais

Os operadores relacionais são utilizados para realizar comparações entre dois valores de um mesmo tipo. Esses valores podem ser representados por variáveis ou constantes. Os operadores relacionais são os seguintes:

DESCRIÇÃO	SÍMBOLO
igual a	=
maior que	>
menor que	<
maior ou igual a	>=
menor ou igual a	<=
diferente de	!=

Tabela 4 – Operadores Relacionais

A uma comparação realizada utilizando um operador relacional se dá o nome de relação. O resultado obtido de uma relação é sempre um valor lógico, ou seja, *verdadeiro* ou *falso*.

Na tabela abaixo temos exemplos de relações e seus resultados. Para tais exemplos, considere duas variáveis inteiras, *A* e *B* onde A=5 e B=8:

RELAÇÃO	RESULTADO
A=B	Falso
A < B	Verdadeiro
A>=B	Falso
B!= 6	Verdadeiro
A >= 5	Verdadeiro

Tabela 5 – Exemplos de relações

2.6.2. Operadores Lógicos

Os operadores lógicos retornam verdadeiro ou falso de acordo com seus operandos. Os operadores lógicos mais comuns são listados na tabela abaixo:

OPERADORES LÓGICOS
E
OU
NÃO

Tabela 6 – Operadores Lógicos

Os operadores lógicos também são conhecidos como conectivos, pois são utilizados para formar novas proposições a partir da junção de duas outras. Para entender o funcionamento de operadores lógicos, vamos recorrer ao nosso exemplo das variáveis inteiras, A e B onde A=5 e B=8:

RELAÇÃO	RESULTADO
A<6 E B > 7	Verdadeiro: o valor de A é menor que 6 **E** o valor de B é maior que 7.
A = 5 E B < 5	Falso: apesar de o valor de A ser igual a 5, o valor de B não é menor que 5.
A = 5 OU B < 5	Verdadeiro: usando o operador **OU,** se ao menos uma das condições for verdadeira (A=5), o resultado da expressão é verdadeiro.

Tabela 7 – Funcionamento dos Operadores Lógicos

Você deve ter notado, pelos exemplos anteriores:

- quando utilizamos o operador lógico **E**, o resultado só será verdadeiro se as **duas** condições relacionadas forem **verdadeiras**;
- para o operador OU, basta que **uma** das condições seja **verdadeira** que o resultado será verdadeiro;
- em consequência: para o operador OU, para que o resultado seja **falso**, as duas condições devem ser **falsas**.

Para visualizar todas as opções possíveis ao utilizar operadores lógicos, utilizamos as *tabelas-verdade*. As tabelas-verdade definem os resultados apresentados pelos operadores lógicos de acordo com todas as combinações possíveis para os valores de suas entradas.

Abaixo são apresentadas as tabelas-verdade para os 3 operadores lógicos que utilizaremos (**OU**, **E** e **NÃO**) para duas proposições (ou expressões) P e Q.

P	Q	P ou Q
V	V	V
V	F	V
F	V	V
F	F	F

Tabela 8 - Tabela Verdade do Operador *OU*

P	Q	P e Q
V	V	V
V	F	F
F	V	F
F	F	F

Tabela 9 - Tabela Verdade do Operador *E*

P	Não P
V	F
F	V

Tabela 10 - Tabela Verdade do Operador *NÃO*

2.6.3. Expressões Lógicas

Como foi dito no início desta seção, as *expressões lógicas* são expressões formadas a partir do uso de variáveis e constantes, operadores relacionais e operadores lógicos. As expressões lógicas são avaliadas e retornam sempre um valor lógico: *verdadeiro* ou *falso*.

As relações utilizadas na tabela 5 para explicar a utilização dos operadores relacionais são bons exemplos de expressões lógicas.

Outros Exemplos:

(x < y) e (y < z)

(y + z < x) ou (x>10) e (y < 5)

Como podemos ver nos exemplos acima, podem ser combinados, em uma mesma expressão, operadores relacionais, lógicos e aritméticos. Assim, é importante compreender a ordem de precedência entre eles, pois isso irá definir a forma de solução da expressão.

PRIORIDADE	OPERADOR
1ª	operadores aritméticos seguindo a ordem de precedência indicada na seção 2.5 (tabela 3)
2ª	operadores relacionais
3ª	operador lógico ***NÃO***
4ª	operador lógico ***E***
5ª	operador lógico ***OU***

Tabela 11 – Ordem de Precedência entre Operadores em Expressões Lógicas

Exemplo: Dadas as variáveis e as seguintes atribuições:

var inteiro NUM1=10;

var inteiro NUM2=5;

var inteiro NUM3=200;

var inteiro NUM4=200;

Vamos verificar se a expressão (NUM1 + NUM2 > 10 **E** NUM1 + NUM3 > NUM4) é VERDADEIRA (**V**) ou FALSA (**F**):

Vamos analisar todas as etapas necessárias:

1. NUM1 + NUM2 > NUM1 é o mesmo que (10 + 5 > 10). Pela tabela de precedências, vimos que primeiro resolvemos os operadores aritméticos; assim temos (15 > 10). Logo, a resposta é **V**, já que 15 é maior que 10.
2. NUM1 + NUM3 > NUM4 é o mesmo que (10 + 200 > 200). Assim, a resposta é **V**, já que 10 + 200 é maior que 200.
3. Assim, nossa expressão se resumirá em **V E V**.

4. Na tabela verdade aprendemos que numa proposição **V** **E** **V,** o resultado será V.
5. Portanto, o resultado final é: **V**, ou seja, Verdadeiro.

Dica!

Hum! Isto pode ficar confuso.

- Uma forma de você evitar confusão é definir as precedências usando parênteses. As expressões dentro dos parênteses mais internos são resolvidas primeiro.
- Você pode testar o uso de parênteses também usando uma planilha de cálculo como o Excel.

Observe as seguintes declarações de variáveis e suas respectivas atribuições e responda às questões abaixo:

var inteiro NUM1 = 10;
var inteiro NUM2 = 5;
var inteiro NUM3 = 200;
var inteiro NUM4 = 200;

16. Coloque F ou V nas expressões abaixo:
Exemplo: (F) NUM4 > NUM3;

a) ( ) NUM1 > NUM2;
b) ( ) NUM1 < NUM3;
c) ( ) NUM1 < NUM4;
d) ( ) NUM3 = NUM4;

17. Coloque F ou V nas expressões abaixo:
Exemplo: (F) NUM1-NUM2 < NUM2;

a) ( ) NUM1 + NUM2 > NUM3;
b) ( ) NUM1 * NUM2 < NUM4;
c) ( ) NUM3 – NUM4 != NUM4;
d) ( ) NUM3 / NUM1 < NUM4;

18. Coloque F ou V nas expressões abaixo:
Exemplo: (F) NUM1+ NUM2 > 10 e NUM3 - NUM4 = NUM3;

a) ( ) NUM1 / NUM2 > 0 **e** NUM1 + NUM3 > NUM4;
b) ( ) NUM1 * NUM2 > 40 **e** NUM3 - NUM1 > NUM4;
c) ( ) NUM1 - NUM2 = 10 **e** NUM2 + NUM3 > NUM4;
d) ( ) NUM1 + NUM2 < 10 **e** NUM3 - NUM4 = NUM1;

19. Coloque F ou V nas expressões abaixo:
Exemplo: (V) NUM3 / NUM2 >55 ou NUM1+ NUM3 > NUM4;

a) ( ) NUM3 / NUM2 > 0 **ou** NUM1 + NUM3 > NUM4;
b) ( ) NUM2 * NUM1 = 50 **ou** NUM3 - NUM1 > NUM4;
c) ( ) NUM1 - NUM2 > 10 **ou** NUM2 + NUM3 > NUM4;
d) ( ) NUM1 + NUM2 > 10 **ou** NUM3 / NUM1 > NUM4;

20. Coloque F ou V nas expressões abaixo:
Exemplo: (V) NUM1 > NUM2 **e** NUM2 < NUM3 **ou** NUM3 < NUM4;

a) ( ) NUM1 > NUM2 **e** NUM2 < NUM3 **ou** NUM3 < NUM4;
b) ( ) NUM1 * NUM2 > 10 **e** NUM1 > NUM4 **ou** NUM3 - NUM1 > NUM4;
c) ( ) NUM1 > 10 **ou** NUM1> NUM4 **e** NUM3 - NUM1 > NUM4;
d) ( ) NUM1 + NUM2 > 10 **ou** NUM1 / NUM3 > NUM4 **e** NUM3 < NUM4;

21. O algoritmo abaixo deverá ler duas notas, calcular a média e mostrar o resultado. Para que o algoritmo seja executado corretamente, complete-o com os comandos que faltam:

Linha 1 ... Algoritmo media
Linha 2 ... **inicio**
Linha 3 ... **var** ____ NOTA1, NOTA2, MÉDIA;
Linha 4 ... _____ (“Digite a primeira nota”);
Linha 5 ... _____ (NOTA1);
Linha 6 ... **escreva** (“_____”);
Linha 7 ... **leia** (_____);
Linha 8 ... MÉDIA ← (_____ + _____)/2;
Linha 9 ... **escreva** (“A média das notas é:”, _____);
Linha 10 ... **fim.**

22. Faça o mesmo no algoritmo abaixo, cuja finalidade é calcular 8% de aumento sobre um salário.

Linha 1 ... Algoritmo reajuste;
Linha 2 ... **inicio**
Linha 3 ... **var** _____ SALARIO, SALARIO_NOVO;

Linha 4 ... _____ (“Digite o salário”);
Linha 5 ... _____ (SALARIO);
Linha 6 ... SALARIO_NOVO ← _____* 1.08;
Linha 7 ... _____ (“O valor do novo salário é:”, _____);
Linha 8 ... **fim.**

23. Faça um algoritmo que leia um número inteiro e imprima seu antecessor e seu sucessor.

24. Faça um algoritmo que leia dois números reais e imprima a soma e a média aritmética desses números.

25. Faça um algoritmo que receba como entrada as medidas dos dois catetos de um triângulo retângulo e calcule e exiba a medida da hipotenusa e a área do triângulo.

2.7. ESTRUTURAS DE SELEÇÃO

Como vimos no capítulo 1, muitas vezes precisamos tomar decisões que podem interferir diretamente no andamento do algoritmo. A representação dessas decisões em nossos programas é feita através do uso de estruturas de seleção, ou estruturas de decisão.

Os **comandos de seleção** ou de **decisão** são técnicas de programação que conduzem a estruturas de programas que não são totalmente sequenciais. Uma estrutura de seleção permite a escolha de um grupo de ações a ser executado quando determinadas condições, representadas por *expressões lógicas*, são ou não satisfeitas.

A estrutura de seleção que utilizaremos em Portugol será a estrutura SE ENTÃO SENÃO. A sintaxe dessa estrutura é exibida a seguir:

Sintaxe: **SE** ***<expressão lógica>***
ENTÃO
<bloco de instruções verdade>
SENÃO
<bloco de instruções falso>
FIM SE;

Como Funciona?

A palavra reservada **SE** indica o início da estrutura de seleção. Após essa palavra, vem a condição que definirá o bloco a ser executado. Qualquer expressão lógica poderá ser utilizada como condição, pois deverá retornar verdadeiro ou falso. Caso a expressão de condição seja verdadeira, o bloco de instruções **ENTÃO** será executado. Caso contrário, o bloco **SENÃO** o será. A palavra reservada **FIM SE** indica o final da estrutura de seleção.

Como primeiro exemplo, faremos um programa que solicita que o usuário preencha suas duas notas, tire a média das duas e, caso a média seja maior ou igual a 60, apresente uma mensagem parabenizando-o pela aprovação:

```
Algoritmo Media
início
    var real nota1; ..... /*criamos a variável real nota 1*/
    var real nota2; ..... /*criamos a variável real nota 2*/
    var real media; ..... /*criamos a variável real media*/
    leia (nota1); ..... /*esperamos que o usuário digite a nota1*/
    leia (nota2); ..... /*esperamos que o usuário digite a nota2*/
    media ← (nota1 + nota2)/2; ..... /*atribuímos o resultado da expressão
                                       que faz a média das notas*/
    se media >= 60 ..... /*verificamos se a média é maior que 60*/
        então
            escreva ("Parabéns! Aprovado!); ..... /*se média maior que
                                                    60 apresenta mensagem de aprovação*/
    fim se; ..... /*finaliza a estrutura de seleção*/
fim.
```

Note, no algoritmo acima, que não utilizamos o bloco SENÃO da estrutura de seleção. Com isso, caso a média seja menor que 60, nada será executado em nosso programa.

Assim, a seguir apresentamos uma estrutura de decisão acrescentando o bloco SENÃO para que apresente uma mensagem de reprovação.

```
início
    /*bloco de declaração e leitura de variáveis igual ao do exemplo anterior*/
    se media >= 60  ..... /*verificamos se a média é maior que 60*/
        então
            escreva ("Parabéns! Aprovado!); ..... /*se média maior que 60 apresenta mensagem de aprovação*/
        senão
            escreva ("Aluno Reprovado!); ..... /*se média menor que 60 apresenta mensagem de reprovação*/
        fim se; ..... /*finaliza a estrutura de seleção*/
    fim.
```

26. O algoritmo abaixo deve ler o salário bruto e calcular o salário líquido. Neste exemplo, o salário líquido será o salário bruto menos os descontos de INSS e IR, seguindo as seguintes regras: caso o salário seja menor que R$ 1.500,00, não devemos descontar IR e descontaremos 8% de INSS; para salários a partir R$ 1.500,00, descontaremos 5% de IR e 11% de INSS. Ao final deve ser exibido o novo salário. Para que o algoritmo seja executado corretamente, complete-o com os comandos que faltam.

***Obs**.: Essas faixas de cálculo são fictícias, apenas para exemplo, não condizendo com as leis em vigor no país.*

```
Linha 1   ... Algoritmo calcula_liquido;
Linha 2   ... inicio
Linha 3   ... var _______ bruto, liquido, inss, ir;
Linha 4   ... _____________ ("Digite o salário");
Linha 5   ... _____ (bruto);
Linha 6   ... se _____________ < 1500
Linha 7   ... então
Linha 8   ... inss ← _____________* 0.08;
Linha 9   ... ir ← _____________;
Linha 10  ... senão
Linha 11  ... ____ ← bruto * 0.11;
Linha 12  ... ir ← bruto * _____;
Linha 13  ... fim se;
Linha 14  ... liquido ← bruto - _____ - _____;
Linha 15  ... _____________ ("O valor do salário
              líquido é:", _________________);
Linha 16  ... fim.
```

27. Sabendo que triângulo é uma figura geométrica de três lados onde cada um dos lados é menor que a soma dos outros dois, queremos fazer um algoritmo que receba três valores e verifique se eles podem ser os comprimentos dos lados de um triângulo. Neste contexto, complete o algoritmo abaixo para que funcione:

Linha 1 ... Algoritmo verifica_triangulo;
Linha 2 ... **inicio**
Linha 3 ... **var** real lado1, lado2, lado3;
Linha 4 ... _____ ("Digite os valores dos 3 lados.");
Linha 5 ... _____ (lado1);
Linha 6 ... _____ (lado2);
Linha 7 ... leia (____);
Linha 8 ... se lado1 + lado2 < _____ e lado2 + lado3 < lado1 _____ lado1 + _____ < lado2
Linha 9 ... então
Linha 10 ... _____ ("Podemos construir um triângulo com estas dimensões!");
Linha 11 ... senão
Linha 11 ... escreva ("_____");
Linha 12 ... fim _____;
Linha 16 ... **fim.**

28. Escreva um algoritmo que leia um número inteiro e diga:
    - Se ele é par ou ímpar. Dica: utilize o operador % (resto da divisão inteira).
    - Se ele é positivo, negativo ou nulo (zero).

29. Escreva um algoritmo que leia a idade de um atleta e escreva na tela em que categoria ele se enquadra, seguindo a tabela abaixo:

Faixa Etária	**Categoria**
de 5 a 10 anos	infantil
de 11 a 17 anos	juvenil
de 18 a 30 anos	profissional
acima de 30 anos	sênior

3. INTRODUÇÃO À LINGUAGEM C

Caro Aluno

Vamos iniciar o terceiro capítulo da nossa disciplina. Agora vamos começar a aplicar os conceitos vistos nos capítulos anteriores em uma linguagem de programação, a linguagem C, e criaremos nossos primeiros programas para computador. Assim, nesta etapa, é importante que as atividades sejam feitas no computador. Desenvolveremos programas que nos ajudarão a melhorar nosso raciocínio lógico e a nossa agilidade na solução de problemas em uma linguagem que o computador é capaz de interpretar.

Se você ainda não fez o download do ambiente Bloodshed Dev-C++, este é o momento!

Apertem os cintos que a nossa viagem está só começando!

3.1. CONCEITOS BÁSICOS

No capítulo anterior construímos nossos algoritmos utilizando uma linguagem conhecida como portugol. Portugol é muito utilizada para iniciar o ensino de programação por ter regras formais e rígidas como uma linguagem de programação e, ao mesmo tempo, ser muito parecida com a linguagem natural entendida pelos humanos.

Mas, quando queremos construir algoritmos que computadores possam entender e executar, é necessário que utilizemos uma linguagem de programação que disponha de um compilador que transforme o algoritmo em um programa a ser executado.

O arquivo contendo o algoritmo que desenvolvemos é chamado de "fonte", pois é a partir dele que o compilador vai criar o programa a ser executado.

Em nosso curso, a linguagem escolhida foi a linguagem C. Para compilar e executar nossos programas, utilizaremos o ambiente Bloodshed Dev-C++, disponível gratuitamente no link http://superdownloads.uol.com.br/download/199/bloodshed-dev-c/.

- **linguagem de programação:** Uma Linguagem de Programação é um método padronizado para expressar instruções para um computador. (LAUREANO, 2005, p. 4).
- **programas:** Um programa de computador é uma coleção de instruções que descrevem uma tarefa a ser realizada por um computador. (LAUREANO, 2005, p. 4).
- **compilador:** programa que traduz algoritmos construídos em uma determinada linguagem de programação para arquivos em linguagem de máquina, ou seja, possíveis de serem executados em computadores.

3.2. CONHECENDO O BLOODSHED DEV-C++

3.2.1. 1º passo – janela 1

Assim que entrarmos no ambiente Dev-C++, a tela abaixo (figura 1) será a primeira a que teremos acesso.

Clique no botão <Fechar> da janela "Dica do dia".

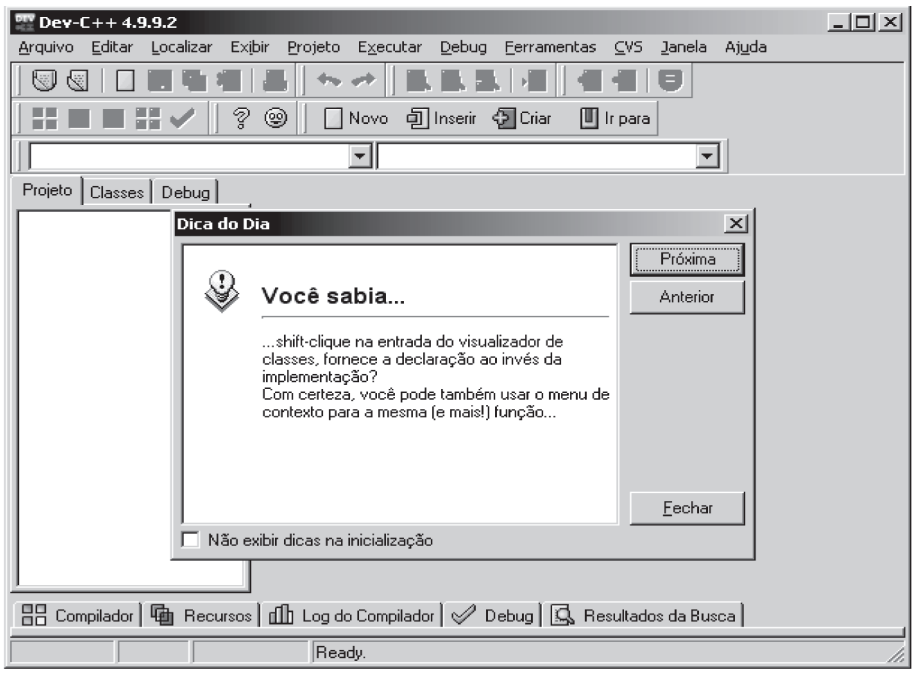


Figura 1: Apresentação da primeira tela

3.2.2. 2º passo – janela 2

Clique no menu Arquivo>Novo>Arquivo Fonte, como apresentado na Figura 2:

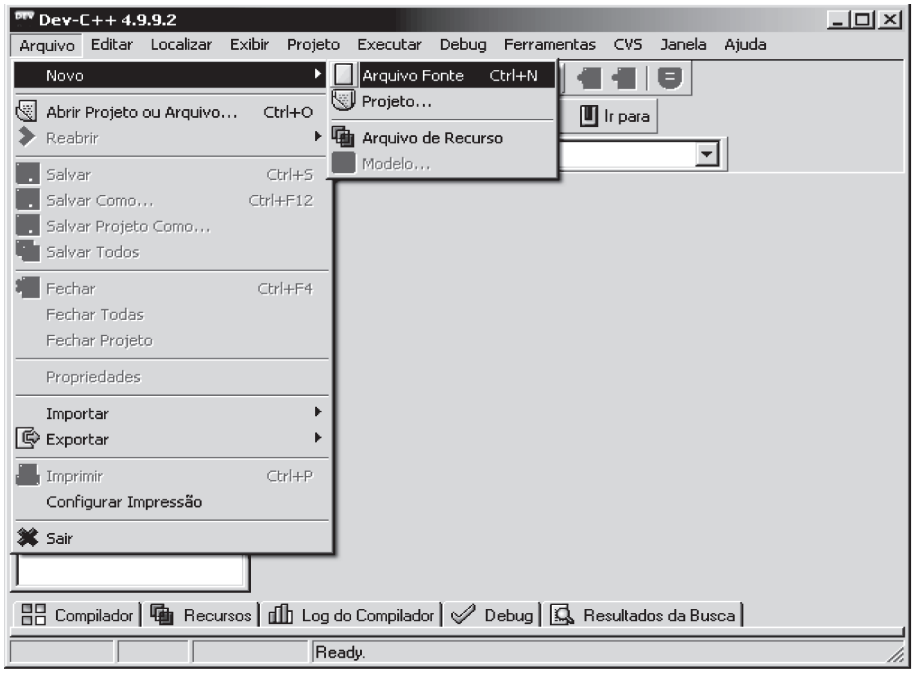


Figura 2: Apresentação do Menu para criação de novo arquivo

3.2.3. 3º passo – janela 3

Será aberta uma janela como a exibida na Figura 3. Digitaremos nossos algoritmos na área branca do lado direito dessa janela, onde aparece o cursor e uma linha destacada em azul.

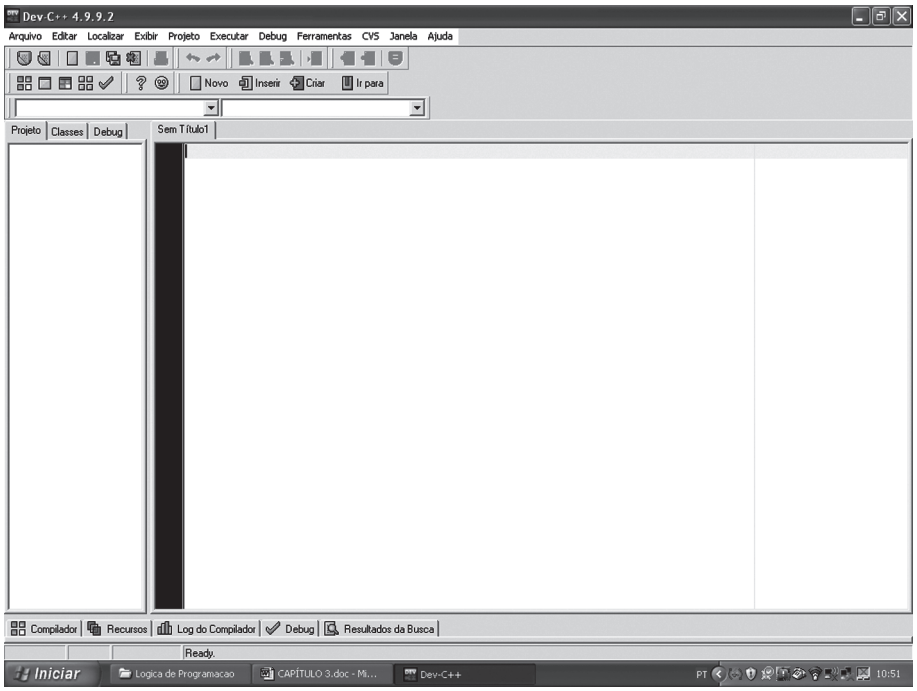


Figura 3: Apresentação da Área de Trabalho

3.3. VISÃO GERAL DA LINGUAGEM C E DA UTILIZAÇÃO DO DEV-C++

Para termos uma visão geral da linguagem que usaremos no desenvolvimento dos programas, vamos analisar como ficaria, na Linguagem C, nosso algoritmo *Multiplicacao* apresentado como exemplo na seção 2.5:

LINGUAGEM – PORTUGOL	LINGUAGEM – C
linha 1 ... Algoritmo multiplicacao;	linha 1 ... #include <stdio.h>
linha 2 ... inicio	linha 2 ... #include <stdlib.h>
linha 3 ... var int NUM1, NUM2, MULT;	linha 3 ... int main ( )
linha 4 ... escreva (“Digite o primeiro número”);	linha 4 ...{
linha 5 ... leia (NUM1);	linha 5 ... int num1, num2, mult;
linha 6 ... escreva (“Digite o segundo número”);	linha 6 ... printf (“Digite o primeiro numero:”);

LINGUAGEM – PORTUGOL	LINGUAGEM – C
linha 7 ... leia (NUM2);	linha 7 ... scanf (“%d”, &num1);
linha 8 ... MULT← NUM1 * NUM2;	linha 8 ... printf (“Digite o segundo numero: ”);
linha 9 ... escreva (“A multiplicação é:”, MULT);	linha 9 ... scanf (“%d”, &num2);
linha 10 ... fim	linha 10 ... mult = num1 * num2;
	linha 11 ... printf (“A multiplicacao e:%d\n”,mult);
	linha 12 ... system (“PAUSE”);
	linha 13 ... return (0);
	linha 14 ...}

Abaixo explicamos cada linha da função *Multiplicacao* em linguagem C, aproveitando para comentar sobre os fundamentos básicos da linguagem C:

- **A primeira linha e a segunda linha -** #include <stdio.h> #include <stdlib.h>

  As duas linhas indicam a inclusão de bibliotecas que possuem as funções de entrada e saída de dados necessárias à execução do nosso programa ***Multiplicacao***. Veremos mais adiante que outras bibliotecas serão necessárias. Quando isso acontecer, vamos incorporá-las. Para evitar problemas, **sempre** inicie seus programas com essas duas linhas.

- **A terceira linha -** int main ( )

  A função main ( ) é sempre a primeira a ser executada no programa C. Em todo programa desenvolvido em C, existirá uma função main ( ).

- **A quarta linha -** {

  É o início de um bloco de comandos no programa. Para toda chave { que inicia um bloco de comandos, teremos uma chave } que será responsável por informar o fechamento desse bloco.

- **A quinta linha -** int num1, num2, soma;

  Foram declaradas as variáveis necessárias à execução do programa. Iniciamos a declaração informando que as variáveis seriam do tipo inteiro (int).

  Observe a existência de um ponto-e-vírgula “;”. Seu emprego indica o final do comando**. Toda instrução em C é finalizada por um ponto-e-vírgula.**

- **A sexta linha –** printf (“Digite o primeiro numero:”);

  A função printf ( ) é uma função de **saída de dados**. Permite que uma mensagem seja exibida no monitor. As mensagens devem ser escritas entre aspas.

- **A sétima linha -** scanf (“%d”, &num1);

  A função scanf ( ) é responsável por **ler os dados** que forem digitados pelo teclado. Nessa linha a função lerá o primeiro número que for digitado e o armazenará no endereço da variável num1, conforme indicado (“%d”, &num1). O “%d” indica que se trata da leitura de um número inteiro. Para ler dados de outros tipos serão utilizados outros códigos, conforme veremos mais à frente.

- **A décima linha -** mult = num1 * num2;

  O comando de atribuição (=) atribui à variável mult o resultado da multiplicação dos valores contidos nos endereços de num1 e num2. **É importante notar que o comando de atribuição que em portugol era representado por uma seta, em C é representado pelo sinal de igual (=)**.

- **A décima primeira linha -** printf (“A soma e: %d \n”, soma);

  Já vimos que a função printf ( ) permite a exibição de mensagens no monitor. Porém, nesse comando o conteúdo da variável *soma* também é exibido. Isso é possível porque incluímos na mensagem o código para impressão de variáveis do tipo inteiro: %d. O código especial \n é responsável por fazer saltar uma linha.

- **A décima segunda –** system (“PAUSE”);

  Possibilita uma pausa no programa a fim de visualizarmos o resultado na tela. Caso contrário, ele seria exibido tão rapidamente que não conseguiríamos vê-lo.

- **A décima terceira linha -** return (0);

  Indica o número inteiro que está sendo retornado pela função, em nosso caso, o número zero. O comando return (0) será detalhado adiante.

- **A décima quarta linha -** }

  Indica o fim do programa. O fim de main ( ).

Um detalhe importante sobre a linguagem C é que, ao contrário de algumas outras linguagens, em C **há distinção entre caracteres maiúsculos e minúsculos**. Assim, em C, é diferente chamar uma variável de *num* ou *Num*. Assim, para evitar erros, por padrão, costumamos utilizar apenas caracteres minúsculos nos nomes de variáveis.

Observe também que todos os comandos da linguagem C são escritos apenas **com caracteres minúsculos**.

Agora que compreendemos cada linha do nosso primeiro programa em C, vamos abrir o ambiente Dev-C++ seguindo os passos apresentados no início do capítulo e, então, digitar esse programa no ambiente.

Para salvar o nosso arquivo fonte devemos acessar o menu Arquivo > Salvar conforme a exibido na Figura 4.

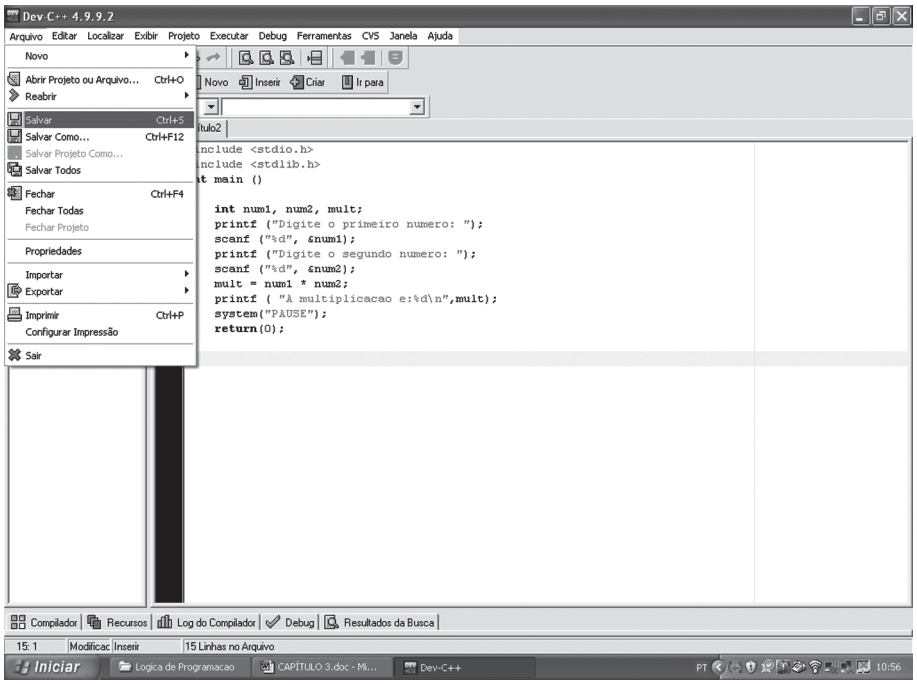


Figura 4: Apresentação do Menu para Salvar arquivo

Então será exibida a janela "Salvar Arquivo". Nessa janela deve ser informado o nome para o arquivo e indicado o tipo do arquivo. No nosso caso devemos salvar como arquivos fontes de C (*C source files*). Essa janela com suas opções é exibida na Figura 5.

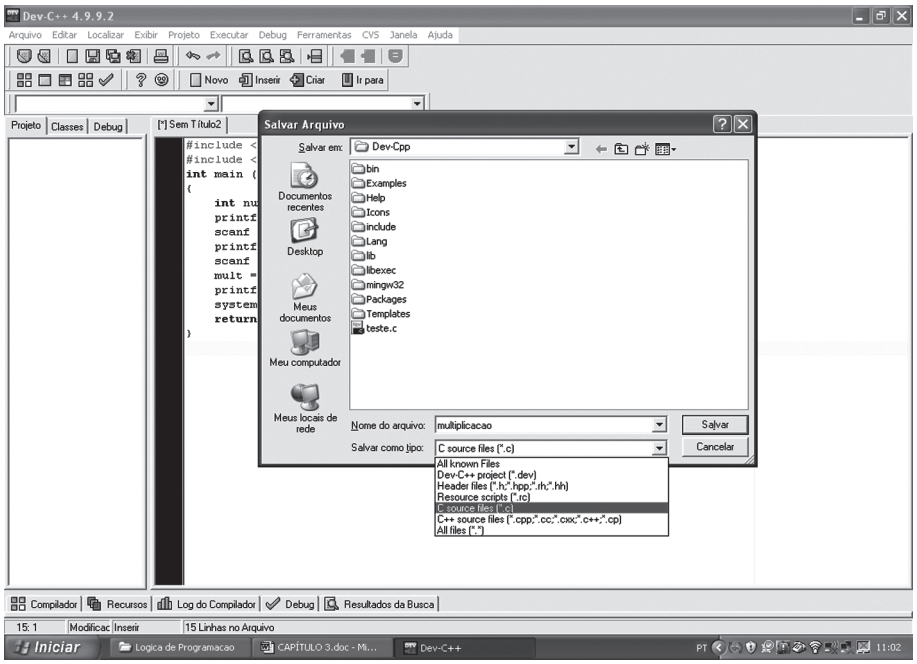


Figura 5: Apresentação da Janela "Salvar Arquivo"

Uma boa prática é salvar periodicamente o arquivo, ou seja, não espere finalizar toda a digitação para então salvar. Assim, caso ocorra algum problema, você não perderá todo o trabalho.

Note que a janela para nomear o arquivo só aparece na primeira vez em que o mesmo é salvo. Nas demais vezes o arquivo será apenas atualizado, não sendo necessário informar novamente seu nome e tipo.

Depois de salvar o arquivo, devemos compilar e executar o programa a fim de visualizarmos seu resultado. Para compilar e executar o programa, podemos utilizar a tecla *F9* ou acessar o menu Executar > Compilar & Executar. Caso você solicite a compilação antes de salvar o arquivo, automaticamente aparecerá a janela de Salvar arquivo para depois o ambiente compilar seu programa. Neste caso siga as instruções dadas anteriormente para salvar arquivo. O resultado da compilação e execução deste programa é exibido na Figura 6. No exemplo apresentado pela figura, o usuário digitou 2 para o valor do primeiro número e 3 para o segundo número.

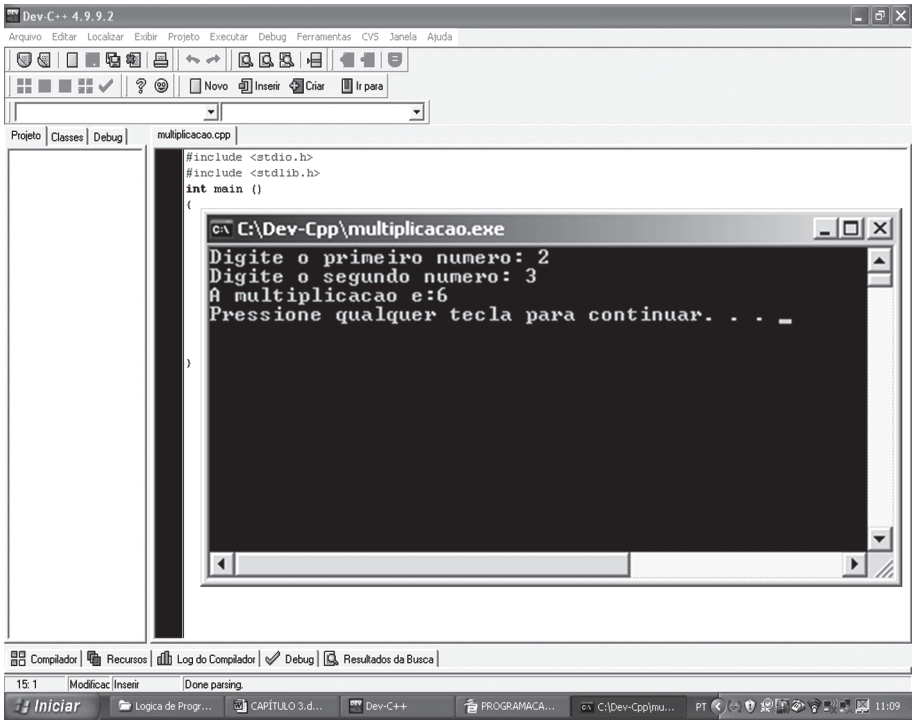


Figura 6: Apresentação do resultado da compilação e execução do programa de Multiplicação

Quando compilamos um programa e o ambiente encontra algum erro no mesmo, a linha que contém o erro fica sombreada em destaque e na parte inferior da janela do ambiente são exibidas mensagens indicando o erro encontrado. Essas mensagens são muito úteis para que possamos compreender o motivo do erro e corrigi-lo. É muito importante ficar atento a tais mensagens. A Figura 7 exibe a tela do ambiente ao tentar executar o programa de multiplicação com um erro.

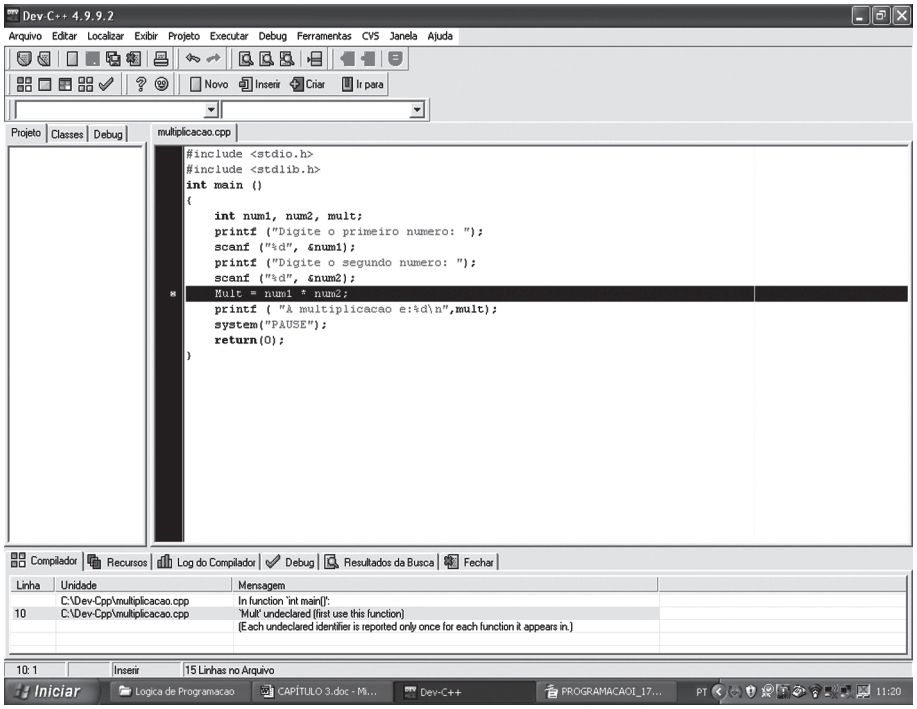


Figura 7: Apresentação de erro de compilação

No caso apresentado pela Figura 7, o erro está no fato de *mult* ter sido declarada com letras minúsculas mas, ao utilizar a variável, colocamos seu nome com a primeira letra maiúscula.

30. Utilizando o ambiente Dev-C++ digite, salve, compile e execute o exemplo do programa de multiplicação conforme apresentado nesta seção.

31. Classifique as afirmativas como verdadeiras ou falsas:

    a) Toda instrução em C é terminada por um ponto-e-vírgula. ( )

    b) Em C não há diferenciação entre letras maiúsculas e minúsculas. ( )

    c) Todo programa C deve ter uma função main ( ). Esta é a primeira função do programa a ser executada. ( )

Olá

As explicações dadas na sequência do conteúdo serão acompanhadas de exemplos que você deverá digitar, compilar e executar no programa DEV-C++.

Depois de executá-los, o código fonte deverá ser analisado e entendido.

A fim de facilitar o estudo, mesmo longe do computador, a partir daqui duas telas serão sempre apresentadas abaixo do exemplo. São elas:

- A tela branca, que contém o código do programa citado como exemplo, devidamente digitado no DEV-C++.
- A tela preta, que é o resultado da compilação e da execução.

Todos os programas desenvolvidos nas atividades também deverão ser digitados, compilados e executados no DEV-C++.

Não avance se as dúvidas permanecerem.

Bom estudo!!

3.4. VARIÁVEIS EM C

Já aprendemos que constantes e variáveis alocam espaço em memória e são utilizadas para armazenar valores necessários à execução do programa. A diferença entre constantes e variáveis está no fato de que o valor de uma constante nunca se altera, enquanto o valor de uma variável pode mudar durante a execução do programa.

A declaração de variáveis em C é muito parecida com a forma que aprendemos em Portugol, ou seja, indicamos o tipo da variável e, em seguida, o nome da mesma. A linha 3 do exemplo apresentado na seção anterior exibe a declaração de 3 variáveis do tipo inteiro. Abaixo são apresentados outros exemplos:

```
int idade;                  //declaração da variável idade do tipo inteiro
float salario, desconto;    //declaração das variáveis salário e desconto do tipo real
```

O tipo da variável define, além do tipo de dado que ela pode armazenar, o tamanho do espaço de memória que deve ser alocado para a mesma. O tamanho do espaço de memória é medido em uma unidade chamada *byte*. Abaixo é apresentada uma tabela que traz os tipos de variáveis existentes em C, informando para cada um o tipo de dados que pode

ser armazenado (fazendo uma comparação com o Portugol) e o tamanho do espaço de memória reservado:

Tipo de Variável em C	Valores a serem armazenados	Tamanho em Bytes
char	Permite armazenar um caractere alfanumérico. Equivalente ao tipo *caractere* de Portugol.	1
int	Permite armazenar números inteiros positivos ou negativos. Equivalente ao tipo *inteiro* de Portugol.	2
float	Permite armazenar valores numéricos reais, ou seja, números com ponto decimal. Equivalente ao tipo *real* de Portugol.	4

Tabela 12: Tipos de variáveis em C

Quanto aos nomes de variáveis, valem as mesmas regras apresentadas quando estudamos Portugol, ou seja, o primeiro caractere do nome deve ser uma letra e os demais podem ser letras, números ou o caractere *underline (_)*.

Vale lembrar que em C há distinção entre caracteres maiúsculos e minúsculos. Assim, caso você declare uma variável com caracteres maiúsculos no nome, e quando for utilizá-la escrever seu nome utilizando apenas caracteres minúsculos, ocorrerá um erro de compilação. Assim, para evitar erros desse tipo, aconselhamos evitar o uso de caracteres maiúsculos nos nomes das variáveis, apesar de seu uso ser permitido.

Após declarar uma variável, pode-se atribuir um valor a ela através da utilização do comando de atribuição *igual (=)*. Em C, a atribuição pode ser feita em qualquer ponto do programa após a criação da variável, mas também é permitido fazer uma atribuição na mesma linha em que é feita a declaração.

Os valores atribuídos a variáveis do tipo *char* devem estar sempre entre aspas. Também é importante ressaltar que o separador decimal utilizado em variáveis do tipo float é o ponto (.) e não a vírgula (,) como costumamos utilizar no Brasil. Assim, se queremos atribuir a uma variável o valor 552,35 devemos utilizar 552.35.

Veja os exemplos:

```
float salario = 552.35;   //a variável salario foi declarada e recebeu o valor 552.35
char sexo = 'F';          //a variável sexo do tipo char foi declarada e recebeu o valor F (note as aspas)
salario = 625.23;         //a variável salario recebeu o valor 625.23
```

3.5. COMANDO DE SAÍDA DE DADOS – *PRINTF ( )*

Como vimos em nosso exemplo do programa *multiplicacao*, a função printf é a função de saída de dados em C. O *printf* funciona em C como a função *escreva* funcionava em Portugol, ou seja, é através dessa função que imprimimos mensagens na tela.

Ainda em nosso exemplo anterior vimos que a função *printf* ( ) usa o caractere de percentual (%) seguido de uma letra para identificar o formato de impressão. Naquele exemplo utilizamos o %d, pois estávamos imprimindo um número inteiro. Na tabela abaixo são exibidos os principais códigos de formatação utilizados no *printf* ( ):

CÓDIGO	SIGNIFICADO
%c	usado quando a função for exibir apenas um caractere (tipo char).
%f	usado quando a função for exibir número com ponto flutuante (tipo float). Exemplo: 1.80
%s	usado quando a função for exibir uma cadeia de caracteres, ou seja, uma ou várias palavras (tipo char[ ]).
%d	usado quando a função for exibir um número inteiro (tipo int).

Tabela 13: Códigos de formato de impressão através do comando *printf*

Abaixo segue um exemplo com *printf* e diferentes tipos de dados:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
     int matricula;
     float media_final;
     char turma;
     turma = 'A';
     matricula = 12;
     media_final = 85.0;
     printf("O aluno matricula = %d \n", matricula);
     printf("Turma = %c \n", turma);
     printf("Ficou com media = %f \n", media_final);
     system("pause");
     return(0);
}
```

Figura 8 – Exemplo com *printf* e diferentes tipos de dados

A *Figura 9* apresenta o resultado da execução desse programa.

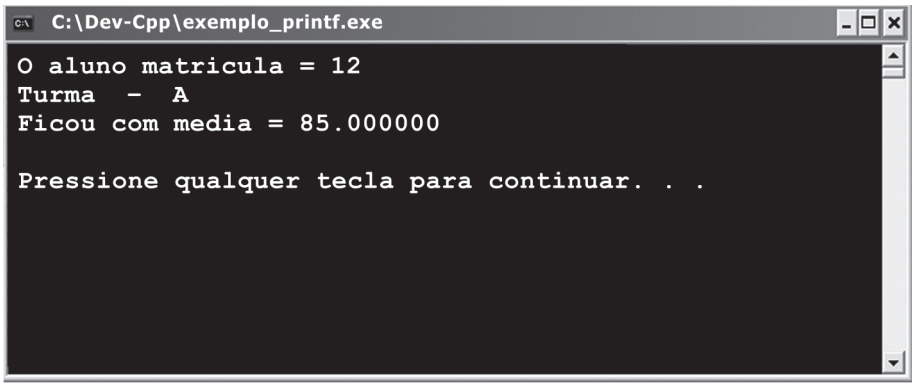


Figura 9 – Exemplo de uso do printf

Nesse exemplo utilizamos o *%d* quando imprimimos a variável *matricula*, que é do tipo *int*, *%c* para a variável *turma*, que é do tipo *char* e *%f* para imprimir a variável *media_final*, do tipo *float*. Os caracteres *\n* que aparecem no final de cada printf são utilizados para pular uma linha; ou seja: caso não tivéssemos utilizado *\n,* todas as mensagens seriam impressas na mesma linha.

3.6. COMANDO DE ENTRADA DE DADOS – *SCANF ( )*

No exemplo do programa *multiplicacao* também pudemos observar a utilização do comando *scanf ( )*. O *scanf ( )* funciona em C como a função *leia* em Portugol, ou seja, é através desta função que lemos entradas de dados através do teclado.

A exemplo do *printf ( ),* o *scanf ( )* também utiliza os códigos de formatação. Enquanto no *printf ( )* esses códigos eram utilizados para indicar o formato dos dados a serem escritos, no *scanf ( )* esses mesmos códigos indicam o formato dos dados a serem lidos. A tabela abaixo exibe os códigos de formatação utilizados no *scanf ( )*. Note a semelhança com a tabela de códigos do *printf ( )*.

CÓDIGO	FUNÇÃO
%c	usado quando a função for armazenar um caractere (tipo char).
%f	usado quando a função for armazenar um número com ponto flutuante, aquele valor com vírgula (tipo float).
%s	usado quando a função for armazenar uma cadeia de caracteres, ou seja, uma ou várias palavras (tipo char[ ]).
%d	usado quando a função for armazenar um número inteiro (tipo int).

Tabela 14: Códigos de formato de leitura através do comando *scanf*

3.7. COMENTÁRIOS

Quando desenvolvemos programas, devemos colocar textos que expliquem o raciocínio seguido durante seu desenvolvimento para que outras pessoas, ou nós mesmos, ao ler o programa mais tarde, não tenhamos dificuldades em entender sua lógica. Esses textos são chamados de comentários.

Os comentários podem aparecer em qualquer lugar do programa. Em C, há dois tipos de comentários: os comentários de linha e os comentários de bloco.

Os comentários de linha são identificados pelo uso de //. Assim, quando usamos // em uma linha, tudo o que estiver nessa linha depois do // são considerados comentários.

Os comentários de bloco são iniciados por /* e finalizados por */. Tudo o que estiver entre esses dois símbolos são considerados comentários. Os comentários de bloco podem ocupar várias linhas.

Veja o exemplo da Figura 10:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int matricula = 2233; /* Podemos escrever comentários desta forma */
    float media_final = 80.5 //ou apenas com duas barras no inicio do comentário
    char discipl[10] = "Prog I"; //a var discipl pode armazenar até 10 caracteres
    printf("O aluno matricula = %d \n", matricula);
    printf("Disciplina = %s \n", discipl);
    printf("Ficou com media = %f \n", media_final);
    system("pause");
    return(0);
```

Figura 10: Exemplo de código em C com comentários

A Figura 11 mostra a execução do programa acima. Note que o comentário só aparece no código fonte, não influenciando na execução do programa.

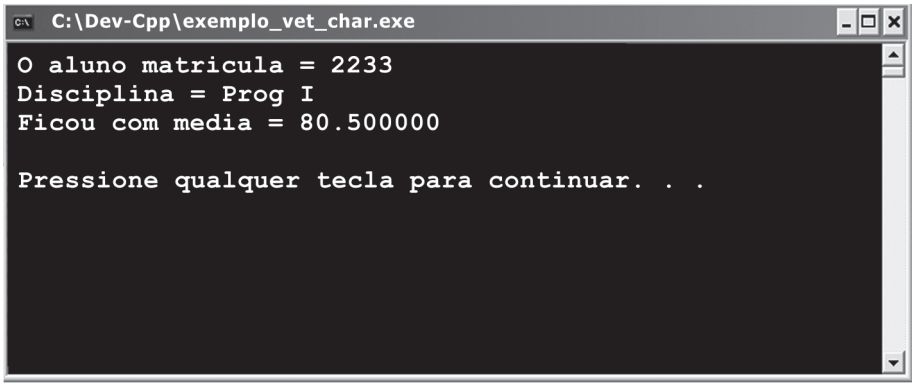


Figura 11: Execução do exemplo de uso de comentários em C

3.8. EXPRESSÕES ARITMÉTICAS

Como estudamos no capítulo 2, os operadores aritméticos são símbolos que representam operações aritméticas, ou seja, as operações matemáticas básicas. A maior parte dos operadores aritméticos de C são os mesmos que vimos em Portugol. Conforme podemos ver na tabela abaixo, apenas acrescentamos o incremento unário (++) e o decremento unário (--):

OPERADOR	OPERAÇÃO MATEMÁTICA
+	Soma
-	Subtração
*	Multiplicação
/	Divisão
- -	Decremento Unário
++	Incremento Unário
%	Resto da Divisão Inteira

Tabela 15: Operadores Aritméticos da linguagem C

O operador de incremento unário (++) incrementa de 1 o seu operando. Ou seja, se eu quiser incrementar em 1 o valor de uma variável x, posso fazer x=x+1; ou fazer simplesmente x++;.

De forma análoga, o operador de decremento unário (--) decrementa de 1 o seu operando. Ou seja, se eu quiser decrementar de 1 o valor de uma variável x, posso fazer x=x-1; ou fazer simplesmente x--;.

Cuidado!

Devemos evitar a utilização de operadores unários em expressões aritméticas, pois seu uso pode dificultar o entendimento da expressão.

Assim, recomendo a utilização desses operadores apenas em ocasiões em que se deseja apenas incrementar ou decrementar o operando; nunca utilizá-los em meio a expressões.

A ordem de precedência entre os operadores em expressões aritméticas é a mesma já estudada, ou seja, primeiro as multiplicações e divisões e só depois as somas e subtrações. Em C também podemos utilizar os parênteses em expressões aritméticas, como fizemos em Portugol.

32. Faça um programa que:

a) peça ao usuário para digitar um número inteiro;
b) armazene esse número numa variável chamada **num1;**
c) peça ao usuário para digitar outro número inteiro;
d) armazene esse número numa variável chamada **num2;**
e) some os valores e guarde o resultado numa variável chamada **soma;**
f) exiba o resultado.

Obs.: *Lembre-se de comentar seu código! Caso encontre dificuldades para fazer esse programa, consulte o exemplo* ***Multiplicacao.***

33. Faça um programa que:

a) peça ao usuário para digitar um número inteiro;

b) armazene esse número numa variável chamada A;

c) peça ao usuário para digitar outro número inteiro;

d) armazene esse número numa variável chamada C;

e) exiba os valores digitados.

Obs.: *Lembre-se de comentar seu código!*

34. Transforme para linguagem C os algoritmos desenvolvidos nos exercícios 23, 24 e 25 (capítulo 2).

4. ESTRUTURAS DE DECISÃO EM LINGUAGEM C

Caro Aluno,

No capítulo anterior desenvolvemos nossos primeiros programas em linguagem C. Os algoritmos desenvolvidos até aqui apresentam estruturas sequenciais, ou seja, todas as instruções do programa são executadas na ordem em que foram escritas.

Mas, como já vimos anteriormente, muitas vezes precisamos que algumas instruções só sejam executadas se alguma condição for atendida.

Para isso, utilizamos comandos de decisão. Neste capítulo conheceremos as estruturas de decisão fornecidas pela linguagem C e desenvolveremos programas utilizando essas estruturas.

Sempre em frente!

4.1 EXPRESSÕES LÓGICAS

Como já estudamos no capítulo 2, as expressões lógicas são expressões formadas a partir do uso de variáveis e constantes, operadores relacionais e operadores lógicos. As expressões lógicas são avaliadas e retornam sempre um valor lógico, em outras palavras, retornam sempre *verdadeiro* ou *falso*.

Revise!

A teoria sobre operadores lógicos, operadores relacionais e tabelas-verdade foi estudada no capítulo 2 e, por isso, não será repetida aqui. Dessa forma, vale a pena revisar tais conteúdos.

Abaixo temos uma tabela que exibe a representação dos operadores lógicos em C:

OPERADOR LÓGICO	REPRESENTAÇÃO EM C
E	&&
OU	|| (duas barras verticais)
NÃO	! (exclamação)

Tabela 16 – Operadores Lógicos em Linguagem C

A tabela 17, a seguir, lista a representação dos operadores relacionais em C:

DESCRIÇÃO	SÍMBOLO
igual a	== (dois sinais de igual)
maior que	>
menor que	<
maior ou igual a	>=
menor ou igual a	<=
diferente de	!=

Tabela 17 – Operadores Relacionais em Linguagem C

Dentre os operadores relacionais, a única alteração que temos em C em relação ao que aprendemos em Portugol refere-se ao operador *igual a*. Em C, esse operador é representado por dois sinais de =, ou seja, por ==. Isso acontece para diferenciar o operador relacional do comando de atribuição.

A Figura 12 exibe um exemplo de utilização do operador relacional = =.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int num1
    num1 = 5; //Atribui o valor 5 à variável num1
    if (num1 == 5) //comparei o valor de num1 com 5. Note que utilizei ==
       printf("num1 vale 5 \n");
    system("pause");
    return(0);
}
```

Figura 12 - Exemplo de utilização do operador relacional ==

A Figura 13 exibe o resultado da execução desse programa.

Figura 13 – Resultado da Execução do Programa Exemplo

4.2. ESTRUTURAS DE SELEÇÃO

Como vimos no capítulo 2, ao desenvolver programas deparamos com situações nas quais o fluxo de execução do programa depende de determinadas condições, ou seja, parte do nosso programa só é executada

se a condição para essa execução for verdadeira. Para isso existem os comandos de seleção ou decisão.

Para realizar essa tomada de decisão na linguagem C temos os comandos de seleção if e switch.

4.2.1. Comando *if*

O comando if deve ser utilizado quando a execução de uma ou mais instruções do programa depender de uma ou mais condições simples. O comando if é equivalente ao comando se..então do Portugol.

Sintaxe: if *(expressão de teste)*
***instrução*;**

Como Funciona?

Se a *expressão de teste* que se encontra entre os parênteses for verdadeira, a *instrução* da linha subsequente será executada; caso contrário, não será.

No caso de termos mais de uma instrução que dependa do resultado da condição para ser executada, essas instruções devem ficar entre chaves, conforme exibido na sintaxe abaixo:

if *(expressão de teste)*
{
***instrução 1*;**
...
***instrução n*;**
}

A figura 14, abaixo, exibe um exemplo em que o resultado da soma de dois números só será exibido se for maior que 2.

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    int n1, n2, s;
    printf("Digite o primeiro numero: ");
    scanf("%d", &n1);
    printf("Digite o segundo numero: ");
    scanf("%d", &n2);
    s = n1 + n2
    if (s > 2) //o comando if verifica se a soma dos números digitados é maior que dois.
       printf("O resultado da soma dos valores digitados é maior do que dois: %d\n", s);
    system("pause");
}
```

Figura 14 – Exemplo de utilização do comando *If*

A Figura 15 exibe o resultado desse programa em um caso em que a soma dos números é maior que 2 e a Figura 16 exibe o resultado da execução quando a soma não é maior que 2.

Figura 15 – Execução do Programa para uma soma maior que 2

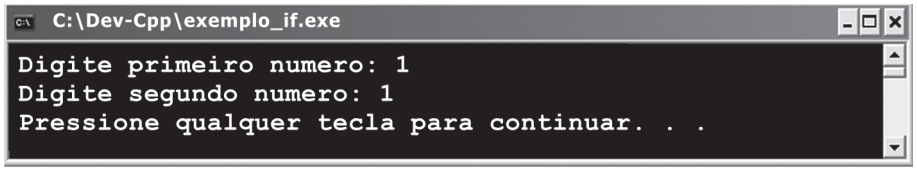


Figura 16 – Execução do Programa para uma soma menor que 2

Observe a Figura 15: o programa não mostrará nada se a soma dos dois números for menor que dois. Apenas será exibida a mensagem **default** "Pressione qualquer tecla para continuar...".

No mais, façamos as atividades propostas a seguir, para verificar como estamos desenvolvendo o nosso aprendizado.

É importante que você digite, compile e execute os exemplos vistos, conforme orientação dada no início deste capítulo.

35. Desenvolva um programa que leia a matrícula e a nota final de cada aluno de uma escola. Se a nota final for maior ou igual a 60, o programa deve exibir a mensagem "Aluno aprovado".

36. Faça um programa que leia o sexo do usuário e apresente a mensagem “O sexo é válido”, se o caractere digitado for ‘M’ ou ‘F’.

37. Faça um programa que leia um número dado como entrada e apresente a mensagem “O número está na faixa de 20 a 90” se o valor fornecido estiver entre 20 e 90.

38. Faça um programa que leia o valor do salário bruto de um funcionário. Se o salário for menor ou igual a R$ 500,00, o programa deve aplicar um aumento de 0.10 (10%).

4.2.2. Comando *if-else*

Como vimos, o comando *if* deve ser utilizado em situações nas quais um bloco de instruções só deve ser executado se uma determinada situação for verdadeira. Mas, muitas vezes deparamos com situações nas quais o programa deve seguir um fluxo caso uma determinada condição seja verdadeira e um outro fluxo caso essa condição seja falsa. Nessas situações, devemos utilizar o comando *if – else*. O comando *if – else* é equivalente ao comando se-então-senão de Portugol estudado no segundo capítulo.

Sintaxe:

```
if (expressão de teste)
{
    < bloco de instruções a ser executado caso a expressão seja verdadeira>
}
else
{
    < bloco de instruções a ser executado caso a expressão seja falsa>
}
```

Como Funciona?

Se a *expressão de teste* que se encontra entre os parênteses for verdadeira, o bloco de instruções logo abaixo do *if* será executado. Caso contrário, o bloco de instruções do *else* é que será executado.

Caso tenhamos apenas uma instrução no bloco do *if* ou no bloco do *else,* as chaves poderão ser omitidas.

Por exemplo, a figura 14 na seção anterior apresenta um programa que exibe uma mensagem caso a soma de dois números seja maior que 2. Caso a soma não atenda a essa condição, nenhuma ação é exercida pelo programa. Agora, vamos alterar aquele exemplo, utilizando o *if – else*. Em nosso novo exemplo, vamos efetuar a soma e, caso a soma seja maior que 2, será exibida uma mensagem informando isso. Caso contrário, será exibida uma mensagem informando que a soma não é maior que 2. Ou seja, vamos apenas acrescentar uma cláusula *else* ao nosso exemplo anterior. O novo exemplo é apresentado na Figura 17.

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

    int num1, num2, s;
    printf("Digite primeiro numero: ");
    scanf("%d", &num1);
    printf("Digite segundo numero: ");
    scanf("%d", &num2);
    s = num1 + num2;
    if (s>2)
        printf("O resultado da soma dos valores digitados e maior que dois: %d \n", s);
    else
        printf("O resultado da soma dos valores digitados nao e maior que dois: %d \n", s);
    system("pause");
    return(0);
}
```

Figura 17 – Exemplo do comando *If...Else*

A Figura 18 exibe o resultado desse programa em um caso onde a soma dos números é maior que 2 e a Figura 19 exibe o resultado da execução quando a soma não é maior que 2.

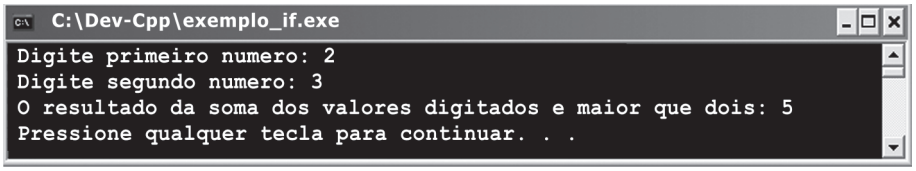


Figura 18 – Execução do Programa para uma soma maior que 2

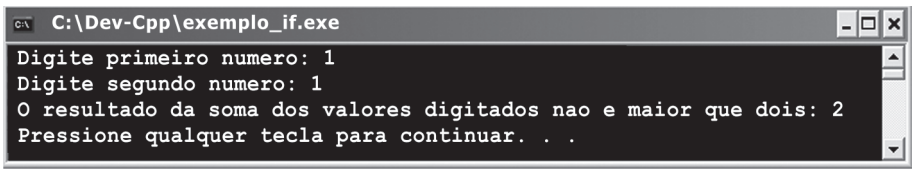


Figura 19 – Execução do Programa para uma soma menor ou igual a 2

Vale ressaltar que, no exemplo anterior, apenas uma instrução é executada tanto no bloco do *if* quanto no bloco do *else*. Por isso, não foi necessário o uso das chaves *{ }*. Caso tivéssemos mais de uma instrução em algum desses blocos, o uso das chaves seria obrigatório!

A Figura 20, abaixo, apresenta um exemplo no qual o uso das chaves foi necessário. Nesse exemplo, uma empresa dará um aumento para os funcionários de acordo com o salário atual de cada um. Caso o funcionário receba até R$ 1.500,00, ele terá um aumento de 10%. Caso o salário seja maior que R$ 1.500,00, o aumento será de 8%. Assim, o programa solicita a digitação do salário e, de acordo com o valor atual, calcula o novo valor e exibe uma mensagem.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    float salario_atual, novo_salario;
    printf("Digite o salario atual: ");
    scanf("%f", &salario_atual);
    if (salario_atual<=1500)
    {
        novo_salario = salario_atual * 1.1;
        printf("Voce tera um aumento de 10%. Seu novo salario sera: %f \n", novo_salario);
    }
    else
    {
        novo_salario = salario_atual * 1.08;
        printf("Voce tera um aumento de 8%. Seu novo salario sera: %f \n", novo_salario);
    }
    system("pause");
    return(0);
}
```

Figura 20 – Exemplo do uso de chaves em comando *If...Else*

A Figura 21 exibe o resultado desse programa em um caso onde o salário é até R$1.500,00 e a Figura 22 exibe o resultado da execução quando o salário atual é maior que R$1.500,00.

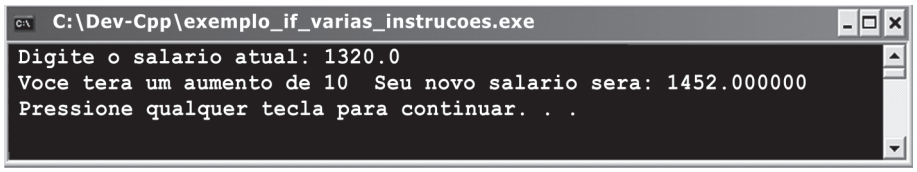


Figura 21 – Resultado da execução para um salário de até R$1.500,00

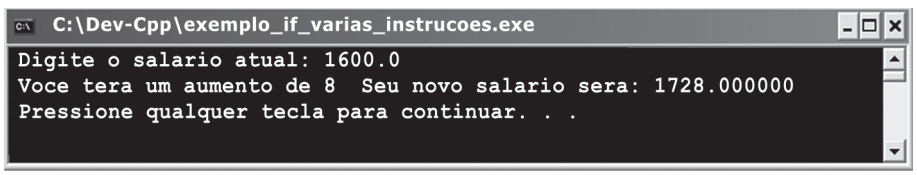


Figura 22 – Resultado da execução para um salário maior que R$1.500,00

Os primeiros exercícios desta lista são complementos aos exercícios da lista anterior. Assim, utilize as soluções da lista anterior como ponto de partida para esta.

39. Como complemento ao exercício 35, o programa deverá exibir também a mensagem "Aluno reprovado", quando a nota final do aluno for menor que 60.

40. Como complemento ao exercício 36, o programa deverá exibir também a mensagem "Sexo inválido", se o caractere digitado for diferente de 'M' ou 'F'.

41. Como complemento ao exercício 37, o programa deverá exibir também a mensagem " O número está fora da faixa de 20 a 90", caso o valor fornecido não esteja entre 20 e 90.

42. Como complemento ao exercício 38, o programa deverá aplicar também um aumento de 0.05 (5%), se o salário for maior do que R$ 500,00.

43. Construa um programa que leia um número inteiro e imprima a informação se este número é ou não divisível por 5. Dica: Utilize o operador % (resto de divisão inteira).

4.1.1. Comandos *if-else* aninhados

Podemos aninhar construções *if-else,* em outras palavras, podemos colocar comandos *if-else* ou comandos *if* dentro de outros comandos *if-else.*

Veja o exemplo apresentado na Figura 23:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    float salario_atual, novo_salario;
    char tem_filhos;
    printf("O funcionario tem filhos: Digite s ou n. ");
    scanf("%c", &tem_filhos);
    printf("Digite o salario atual: ");
    scanf("%f", &salario_atual);
        if (salario_atual<=1500)
    {
        novo_salario = salario_atual * 1.1; //Se salario ate 1500 aumento de 10%
        if (tem_filhos == 's')
           novo_salario = salario_atual + 801; //Se salario ate 1500 e tem filhos aumenta R$80,00
    }
    else
   {
        novo_salario = salario_atual * 1.08; //Se salario ate 1500 aumento de 8%
        if (tem_filhos == 's')
           novo_salario = salario_atual + 50; //Se salario ate 1500 e tem filhos aumenta R$50,00
    }
    printf("Seu novo salario sera: %f \n", novo_salario);
    system("pause");
    return(0);
}
```

Figura 23 – Exemplo de comandos *if-else* aninhados

Nesse exemplo, além do aumento percentual sobre os salários, os empregados também receberão uma ajuda caso tenham filhos. Assim, os empregados com salários até R$ 1.500,00 receberão o aumento de 10% e, se tiverem filhos, receberão mais R$ 80,00. Já os funcionários com salários maiores que R$ 1.500,00 receberão o aumento de 8% e, se tiverem filhos, receberão mais R$50,00. Assim, além de informar o salário, deverá ser informado também se o funcionário tem filho ou não, digitando 's' para sim e 'n' para não. Note que foi acrescentada uma condição *if* dentro dos blocos *if-else* existentes para somar a gratificação no caso de ter filho. Note também que, nesse exemplo, utilizamos um printf único, fora das estruturas de condição que exibe o salário final.

A Figura 24 exibe o resultado da execução deste programa para um funcionário cujo salário é R$ 1.300,00 e que tem filhos.

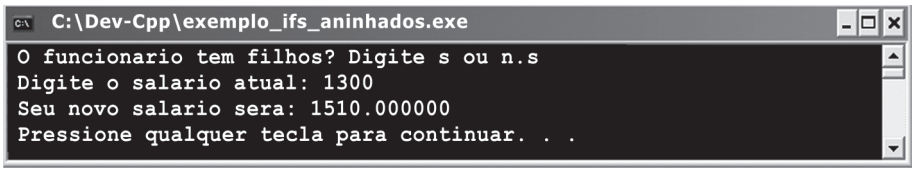


Figura 24 – Resultado da execução para um salário de R$ 1.300,00 e que tem filhos.

Teste mais!

Crie, compile e execute esse programa testando outros valores de salário, variando a resposta à pergunta se tem ou não filhos.

Teste sempre seus programas com vários valores ou várias situações diferentes para poder ter mais segurança.

44. Faça um programa que leia três valores distintos a serem digitados pelo usuário, determine e exiba o menor deles.

45. Sabendo que triângulo é uma figura geométrica de três lados em que cada um dos lados é menor que a soma dos outros dois, faça um algoritmo que receba três valores e verifique se eles podem ser os comprimentos dos lados de um triângulo.

46. Refaça, agora em linguagem C, o algoritmo desenvolvido no exercício 28 (capítulo 2).

47. Faça um programa que leia o salário bruto e calcule o salário líquido. Para esse programa, o salário líquido será o salário bruto menos os descontos de INSS e IR, seguindo as regras:
    - caso o salário seja menor que R$1.500,00, não devemos descontar IR e descontaremos 8% de INSS;
    - para salários a partir R$1.500,00, descontaremos 5% de IR e 11% de INSS.

Obs.: Essas faixas de cálculo são fictícias, apenas para exemplo, não condizendo com as leis em vigor no país.

4.1.2. Comando *switch*

Assim como o comando *if,* o comando *switch* é uma estrutura de decisão. Mas, devemos utilizar o comando *switch* quando o programa deve escolher uma entre várias alternativas para um determinado valor.

Sintaxe:

```
switch (condição de teste)
{
    case constante 1:
        bloco de instruções 1
        break;
    case constante n:
        bloco de instruções n
        break;
    default: bloclo de instruções padrão.
}
```

No exemplo apresentado pela figura 25 utilizamos a estrutura *switch* para simular uma calculadora de quatro operações. Solicitamos a digitação dos dois números e da operação desejada e, após, utilizamos um switch de forma que, de acordo com a operação, imprimimos o resultado. O bloco default só será executado caso seja digitado um valor inválido para o operador, ou seja, se não for +, -, * ou /.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    float num1, num2;
    char operador;

    printf("Digite o primeiro numero, o operador e o segundo numero\n");
    scanf("%f %c %f", &num1, &operador, &num2);
    switch (operador) {
           case '+':
                printf("O resultado e %f\n", num1+num2);
                break;
           case '-':
                printf("O resultado e %f\n", num1+num2);
                break;
           case '*':
                printf("O resultado e %f\n", num1+num2);
                break;
           case '/':
                printf("O resultado e %f\n", num1+num2);
                break;
           default:
                printf("Operador invalido\n", num1+num2);
    }
    system("pause");
    return(0);
}
```

Figura 25 – Exemplo de utilização do comando *switch*

A Figura 26 exibe o resultado da execução desse programa, tendo como entrada os valores 20.5 * 3.

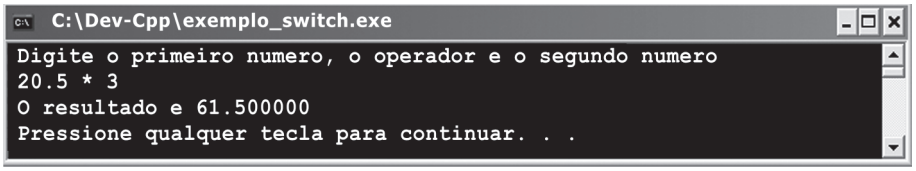


Figura 26 – Resultado de uma execução do programa exemplo

48. Uma empresa dará aumento aos seus funcionários, de acordo com sua Classe:

    a) Classe A = 0,10 (10%) de aumento;

    b) Classe B = 0,15 (15%) de aumento;

    c) Classe C = 0,20 (20%) de aumento.

Usando o ***comando switch***, faça um programa que leia o salário e a classe do funcionário, calcule e exiba os salários com os devidos aumentos.

49. Precisamos fazer um algoritmo para uma biblioteca que receba o tipo do usuário e a classificação do livro e responda se o usuário pode ou não locar o livro seguindo as seguintes regras: Existem dois tipos de usuários: o tipo 'A' (aluno) e o tipo 'P' (professor). Existem duas classificações de livros: A e B. Livros do tipo A podem ser locados por qualquer usuário enquanto livros do tipo B só podem ser locados por professores.

O que aprendemos até aqui?

- Que há três comandos de seleção em C.
- Que o comando ***if*** é utilizado para decisão simples.
- Que o comando ***if-else*** é utilizado quando, com base em uma condição, o programa pode executar um ou outro bloco de comandos.
- Que podemos utilizar comandos if-else ***aninhados***, ou seja, dentro de um bloco de comandos executados em um if-else podemos ter outra estrutura if-else e assim sucessivamente.
- Que o comando ***switch*** é utilizado quando temos condições que não sejam expressões e temos uma lista de valores possíveis para a condição.

5. ESTRUTURAS DE REPETIÇÃO EM LINGUAGEM C

Caro aluno,

No capítulo anterior aprendemos a utilizar estruturas de decisão em linguagem C. Mas, como já vimos anteriormente, muitas vezes precisamos que algumas instruções sejam executadas repetidamente até que alguma condição seja atendida. Para isso, utilizamos comandos de repetição.

Neste capítulo, conheceremos as estruturas de repetição fornecidas pela linguagem C e desenvolveremos programas utilizando essas estruturas.

Vamos em frente!

5.1. ESTRUTURAS DE REPETIÇÃO

Conforme vimos no capítulo 1, são normais situações nas quais nós repetimos determinadas ações enquanto não atingimos um objetivo. Da mesma forma, ao desenvolver nossos programas, deparamos com situações nas quais precisamos que um determinado bloco de instruções seja repetido enquanto uma determinada condição é válida. Nessas situações, utilizaremos os comandos de repetição, também conhecidos como laços ou *loops*.

A linguagem C conta com 3 comandos de repetição: *for, while* e *do while.*

5.1.1. Comando *for*

O comando for é ideal para situações nas quais um bloco de instruções deve ser repetido um número fixo ou conhecido de vezes.

Sintaxe:

for ***(inicialização;teste;incremento)***

{

    bloco de instruções

}

Como funciona:

Os parênteses que seguem a palavra *for* contêm 3 expressões separadas por ponto-e-vírgula: expressão de inicialização, expressão de teste e expressão de incremento.

A expressão de **inicialização** é uma instrução de atribuição executada apenas uma vez, no início do laço. É geralmente utilizada para inicializar uma variável que irá controlar o número de repetições do laço.

A expressão de **teste** é a condição que controla o laço. Normalmente é uma expressão lógica que utiliza a variável de controle do laço. Essa expressão é verificada antes da execução do laço. Se for verdadeira, o laço é executado mais uma vez. Caso contrário, o laço é finalizado.

A expressão de **incremento** define a maneira como a variável de controle do laço será alterada a cada vez que o laço for repetido. Ela é executada ao final da execução de cada repetição do corpo do laço.

Vamos fazer um programa que leia a nota de 10 alunos e no final exiba a média da turma.

```
Linha 1   ... #include <stdio.h>
Linha 2   ... #include <stdlib.h>
Linha 3   ... int main (   )
Linha 4   ...{
Linha 5   ... float nota, soma=0, media;
Linha 6   ... int conta;
Linha 7   ... for (conta=0;conta<=9;conta++)
Linha 8   ... {
Linha 9   ...       printf ( “Digite a nota “);
Linha 10 ...       scanf (“%f”,&nota);
Linha 11 ...       soma=soma+nota;
Linha 12 ... }// esta chave encerra o comando de repetição for
Linha 13 ... media= soma/conta;
Linha 14 ... printf ( “A media da turma e %f \n “, media);
Linha 15 ... system (“PAUSE”);
Linha 16 ... return 0;
Linha 17 ... }
```

Vamos entender melhor algumas linhas do código acima.

linha 5 ... **float** nota, soma=0, media;

Houve necessidade de iniciarmos a variável *soma* com zero, pois terá valor cumulativo.

Já vimos que, ao declararmos uma variável, estamos reservando um espaço na memória, o qual não é necessariamente um espaço limpo. Isso significa que nossa variável no momento da declaração armazena apenas lixo. Ao atribuirmos o valor zero para ela, garantimos que os valores sejam acumulados corretamente.

linha 7 ... **for** (conta=0; conta<=; conta++)

A linha do comando ***for*** controla a quantidade de vezes que o *loop* será executado. Observe que ele inicia a variável **conta** de zero (conta=0;), controla o loop para ser executado 10 vezes (conta<=9) e finalmente incrementa a variável **conta (**conta++).

É importante notar que o comando **conta ++** é o mesmo que: **conta = conta + 1.**

linha 11 ... soma=soma+nota;

Nessa linha acumula-se a soma das notas da turma.

linha 13 ... media= soma/conta;

Observe que essa linha de comando foi colocada após encerramento do ***for***, pois só nos interessa calcular a média depois que todas as notas forem somadas.

Como a variável **conta** guarda o número de vezes que o loop foi executado, que é igual à quantidade de alunos estipulada no programa, em vez de dividirmos a soma por 10, fazemos a divisão utilizando a variável conta.

linha 14 ... **printf** ("A media da turma e %f \n", media);

Para melhorarmos a exibição dessa mensagem, basta trocar %f por %.2f: serão exibidas apenas 2 casas depois da vírgula.

A Figura 27 apresenta nosso programa e o resultado de uma execução do mesmo.

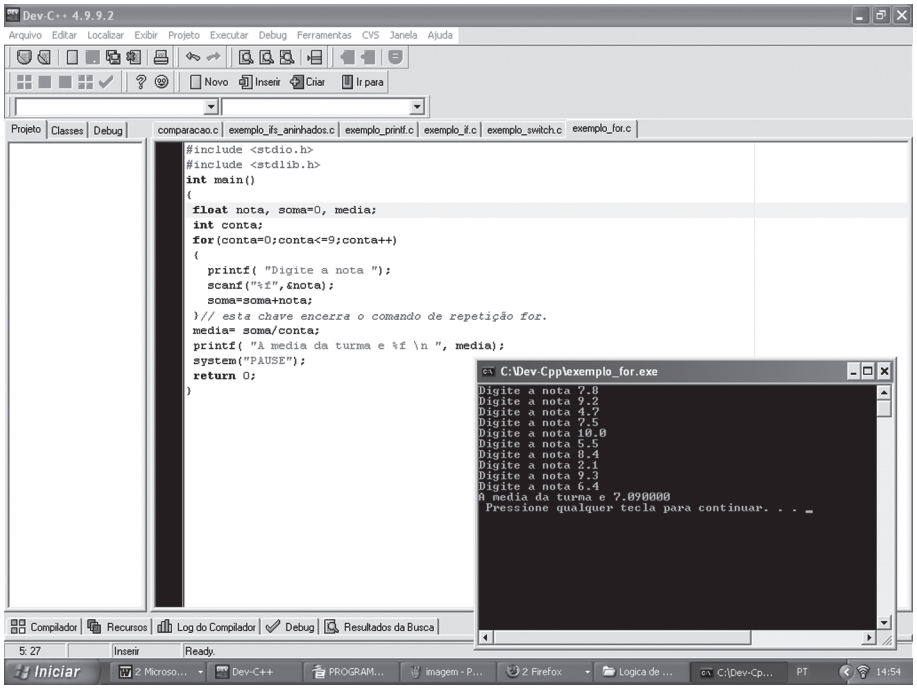


Figura 27 – Código e Resultado da execução do programa exemplo

50. Faça um programa que leia 5 valores reais e imprima o quadrado de cada um deles. Ao fim, imprima também o somatório dos cinco.

51. Faça um programa que calcule a média de 5 números inteiros dados como entrada e imprima o resultado.

52. Faça um programa que imprima todos os números pares no intervalo de 1 a 100.

53. Faça um programa que receba como entrada um valor inicial e um final de temperatura em graus Celsius e imprima, variando do valor inicial até o final, a temperatura em Celsius seguido do seu equivalente em Fahrenheit. Ou seja, faça uma conversão entre as duas medidas. Vale lembrar que a fórmula de conversão de Celsius para Fahrenheit é dada por: F=1,8C + 32, onde F é a temperatura em Fahrenheit e C a temperatura em Celsius. Por exemplo, suponha que o programa receba 8 como inicial e 10 como temperatura final. Ele deve imprimir:

    8 Celsius = 46,4 Fahrenheit

    9 Celsius = 48,2 Fahrenheit

    10 Celsius = 50,0 Fahrenheit

54. Na matemática, o fatorial de um número natural n é dado pelo produto de todos os números inteiros e positivos menores ou iguais a n. Por exemplo, o fatorial de 5 é dado por 5 * 4 * 3 * 2 * 1. Desenvolva um programa que calcule o fatorial de um número dado como entrada.

5.1.2. Comando while

O comando *while* é ideal para situações nas quais não sabemos o número exato de vezes que o bloco de instruções deve ser repetido, mas também pode ser utilizado para substituir laços *for*.

Sintaxe:

while *(condição)*

{

bloco de instruções

}

Como funciona:

Enquanto a condição especificada no cabeçalho do laço for satisfeita, o bloco de instruções é executado. Assim, antes de cada execução do bloco a condição é avaliada. Caso seja verdadeira, o bloco é executado. Caso a condição seja falsa, o laço é finalizado.

Vamos utilizar o mesmo exemplo do comando ***for,*** porém, ao invés de pré-definir que serão entradas dez notas, leremos a primeira nota e, daí em diante questionaremos ao usuário se deseja digitar mais notas.

Nosso código ficará assim:

```
Linha 1   ...#include <stdio.h>
Linha 2   ...#include <stdlib.h>
Linha 3   ...int main (   )
Linha 4   ...{
Linha 5   ... float nota, soma=0, media;
Linha 6   ... int resp=1, contador=0;
Linha 7   ... while (resp==1)
Linha 8   ... { // esta chave inicia o comando de repetição while
Linha 9   ...       printf ( “Digite a nota “);
Linha 10  ...       scanf (“%f”,&nota);
Linha 11  ...       soma=soma+nota;
Linha 12  ...       printf (“Digite 1 para continuar ou digite outra
                    tecla para finalizar.... “);
Linha 13  ...       scanf (“%d”,&resp);
Linha 14  ...       contador++; // Essa linha é igual a
                    contador=contador + 1
Linha 15  ... }
Linha 16  ... media= soma/contador;
```

Linha 17 ... printf (“A media da turma e %.2f \n “, media);

Linha 18 ... system (“PAUSE”);

Linha 19 ... return 0;

Linha 20 ...}

Vamos entender melhor algumas linhas do código acima:

linha 6 ... int resp=1, contador=0;

A variável ***resp*** será responsável por armazenar a resposta do usuário. Perceba que ela é inicializada = 1. Isso se faz necessário para que o programa execute o laço a primeira vez.

A variável ***contador*** guardará a quantidade de vezes em que o usuário digitou uma nota, o que corresponderá à quantidade de alunos. Precisaremos desse total para calcular a média da turma.

linha 7 ... while (resp==1)

Observe que enquanto a **resp** for igual a 1 (um) o laço será executado. Assim podemos entender o motivo pelo qual iniciamos a variável **resp** no momento da sua declaração. Se o valor 1 não fosse atribuído à variável no início, o laço nunca seria executado.

linha 14 ... contador++

A variável contador está contando a quantidade de notas entradas.

A Figura 28 apresenta nosso programa e o resultado de uma execução na qual foram digitadas seis notas.

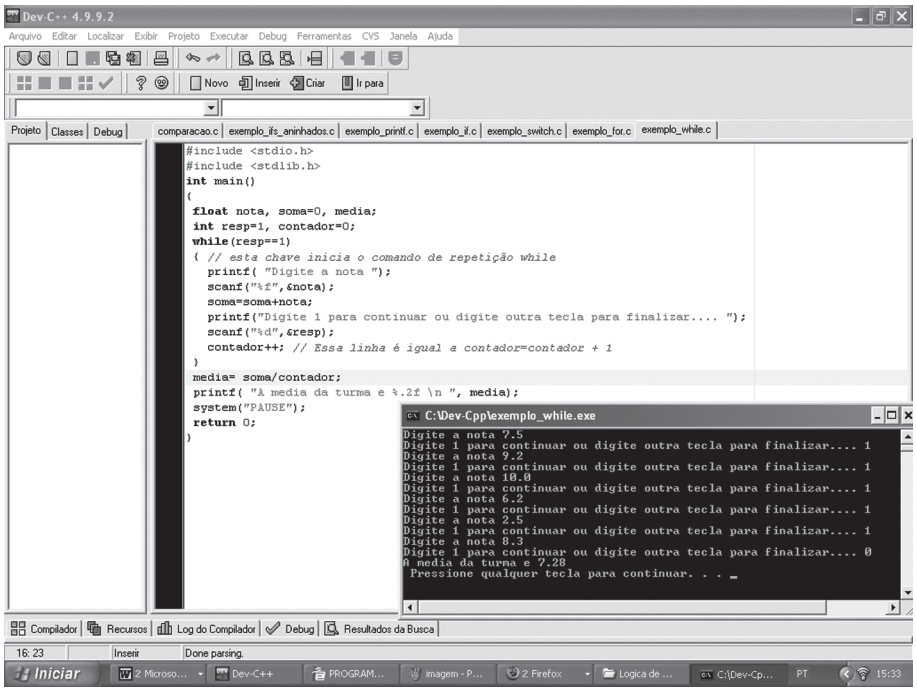


Figura 28 – Código e Resultado da execução do programa exemplo com while

55. Faça um programa que leia os valores dos salários atuais dos funcionários de uma empresa e imprima os valores com aumento. Se o salário for menor ou igual a R$ 500,00, o programa deve aplicar um aumento de 0.10 (10%) e se for maior que R$ 500,00, o aumento deve ser de 0.08 (8%). Assim como no nosso último exemplo, o usuário é que deve informar quando deseja sair. Ou seja, ele deve digitar o salário de um funcionário e o programa vai exibir o valor com o aumento. Depois ele deve responder se deseja digitar outro salário ou não. Se ele responder que sim, deve-se solicitar o novo salário e exibir o valor com aumento. Então torna-se a perguntar se ele deseja informar um novo salário e repetir o laço até que ele não queira informar novos salários.

56. Faça um programa que fique em um laço solicitando a digitação de números inteiros e só pare de solicitar a digitação de novos números quando o usuário informar o número 0. Quando o número 0 for informado, o programa deve exibir a quantidade de números digitados, a quantidade de números pares, a quantidade de números ímpares e a média dos valores dos números digitados.

57. No exercício 54, fizemos um programa para calcular o fatorial de um dado número. Refaça tal exercício utilizando um laço *while* em lugar do laço *for*.

5.1.3. Comando do while

O comando *do while* é muito parecido com o comando *while* que acabamos de aprender. A única diferença é que com o comando *do while* asseguramos que o bloco de instruções do laço seja executado ao menos uma vez. Depois da primeira execução, o bloco de instruções só é executado novamente se a condição for verdadeira.

Sintaxe:

```
do
{
    bloco de instruções
} while (condição)
```

Como funciona:

A primeira execução do bloco de instruções ocorre sem necessitar da avaliação da condição. Depois da primeira execução a condição é avaliada e o bloco de instruções só é executado novamente enquanto a condição for verdadeira.

Para mostrar na prática a utilização do *do while*, vamos refazer o mesmo exemplo utilizado para o laço *while*. Nosso programa ficará assim:

```
Linha 1  ... #include <stdio.h>
Linha 2  ... #include <stdlib.h>
Linha 3  ... int main (  )
Linha 4  ... {
Linha 5  ... float nota, soma=0, media;
Linha 6  ... int resp, contador=0;
Linha 7  ... do
Linha 8  ... {
Linha 9  ...      printf ( “Digite a nota “);
Linha 10 ...      scanf (“%f”,&nota);
Linha 11 ...      soma=soma+nota;
Linha 12 ...      printf (“Digite 1 para continuar ou digite outra
                  tecla para finalizar.... “);
Linha 13 ...      scanf (“%d”,&resp);
Linha 14 ...      contador++; // Esse comando é igual a
                  contador=contador + 1;
Linha 15 ... } while (resp==1);
Linha 16 ... media= soma/contador;
```

Linha 17 ... printf (“A media da turma e %.2f \n “, media);

Linha 18 ... system (“PAUSE”);

Linha 19 ... return 0;

Linha 20 ... }

Vamos entender melhor algumas linhas do código acima:

linha 6 ... **int resp, contador=0;**

Note que nesse exemplo não precisamos iniciar a variável *resp* com 1, pois a condição só será testada após a primeira execução. Assim, na primeira vez em que a condição for testada, o usuário já terá respondido à pergunta.

linha 7 ... **do**

Início do comando *do-while*. O bloco será executado ao menos uma vez.

linha 8 ... **{**

Delimita o início do bloco de instruções do laço.

linha 15 ... } while (resp==1);

Enquanto essa condição for verdadeira, o programa continuará em execução. Delimita o fim do bloco de instruções do laço.

58. Construa um programa capaz de ler uma série de números até que apareça um número entre 1 e 5. Ao final, exiba a quantidade de números digitados, o valor da soma dos números digitados e a média dos valores dos números digitados.

59. Refaça o exercício 55; porém, agora, utilize **do while** em lugar do **while** utilizado anteriormente.

O que aprendemos até aqui?

- Os comandos for, while e do while são responsáveis por repetições do programa.
- Os três comandos podem ser usados para resolver o mesmo problema. Cabe ao programador decidir qual deles melhor responderá às necessidades para a solução de cada problema.

Leituras complementares:

MIZRAHI, Victorine V. **Treinamento em Linguagem C – Curso Completo – Módulo 1**, Mc Graw Hill, 1990.

KERNIGHAN Brian W. **C Linguagem de Programação Padrão ANSI**. Rio de Janeiro: Elsevier, 1989.

6. VETORES E MATRIZES

Caro Aluno,

Até o momento, cada variável que declaramos em nossos programas é capaz de armazenar apenas um valor. Assim, para cada valor que precisamos manipular, nós declaramos uma variável distinta.

Mas existem situações em que são necessárias várias variáveis de um mesmo tipo para armazenar dados relacionados. Nessas situações, utilizaremos os conceitos de vetores e matrizes, os quais serão abordados neste capítulo.

A melhor forma de aprender programação é programando! Não deixe de realizar todas as atividades!

6.1. VETORES

Um vetor é uma estrutura de dados utilizada para representar certa quantidade de variáveis de valores homogêneos, ou seja: um conjunto de variáveis, todas do mesmo tipo.

No capítulo anterior, fizemos um programa que calculava a média de notas de uma turma. Nesse programa sempre líamos a nota de cada aluno utilizando a mesma variável para armazenar o valor. Ou seja, a cada rodada do laço, a variável assumia a nota de um aluno e essa nota era acumulada numa variável *soma*.

Assim, se no final do programa quiséssemos exibir as notas de todos os alunos, não seria possível, pois sempre substituíamos a nota de um aluno pela nota do aluno seguinte. Para que fosse possível armazenar as notas de todos os alunos, teríamos duas alternativas: a primeira, menos inteligente, seria declarar uma variável para armazenar a nota de cada aluno; e a segunda seria declarar um vetor de tamanho igual à quantidade de alunos.

Sintaxe para declaração de vetores:
tipo_do_vetor nome_do_vetor[tamanho];

Como funciona:
A única diferença entre a declaração de vetores e a declaração de variáveis simples é que, ao declarar vetores, especificamos ao fim da declaração e entre colchetes [ ] o tamanho do vetor, ou seja, sua capacidade de armazenamento.

Por exemplo: se precisarmos armazenar as notas de 10 alunos, podemos declarar um vetor de 10 posições do tipo float. A declaração do nosso vetor ficaria assim: ***float notas[10];.***

6.1.1. Referenciando elementos e armazenando dados

Um vetor é como uma coleção de caixinhas enumeradas. Cada caixinha é capaz de armazenar um valor e tem o seu "endereço". O "endereço" de cada caixinha é conhecido como índice e serve para identificar qual posição do vetor queremos acessar. Abaixo temos a representação gráfica no nosso vetor de notas de dez posições declarado acima:

Índice	0	1	2	3	4	5	6	7	8	9
Valor	8.5	7.6	9.3	10.0	6.3	4.7	8.8	9.1	3.4	10.0

No vetor representado acima temos, por exemplo, armazenado no índice "2" o valor "9.3". Note que, como declaramos um vetor com 10 posições, os índices variam de 0 a 9.

Mas, resta uma dúvida: como referenciar as posições do vetor e armazenar dados nas mesmas? A sintaxe para utilizar uma posição do vetor é: ***nome_do_vetor[indice];***

Assim, para armazenar o valor 8.5 na primeira posição do nosso vetor, seria utilizado o comando: ***notas[0] = 8.5;***

Abaixo, a figura 29 apresenta um exemplo simples que utiliza um vetor de 5 posições e atribui um valor a cada posição. A figura 30 exibe o resultado da sua execução.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int numeros[5];
    numeros[0] = 10;
    numeros[1] = 20;
    numeros[2] = 30;
    numeros[3] = 40;
    numeros[4] = 50;
    printf("O valor %d foi armazenado na posição zero do vetor.\n"), numeros[0]);
    printf("O valor %d foi armazenado na posição um do vetor.\n"), numeros[1]);
    printf("O valor %d foi armazenado na posição dois do vetor.\n"), numeros[2]);
    printf("O valor %d foi armazenado na posição tres do vetor.\n"), numeros[3]);
    printf("O valor %d foi armazenado na posição quatro do vetor.\n"), numeros[4]);
    system("pause");
}
```

Figura 29 – Exemplo de programa utilizando Vetor

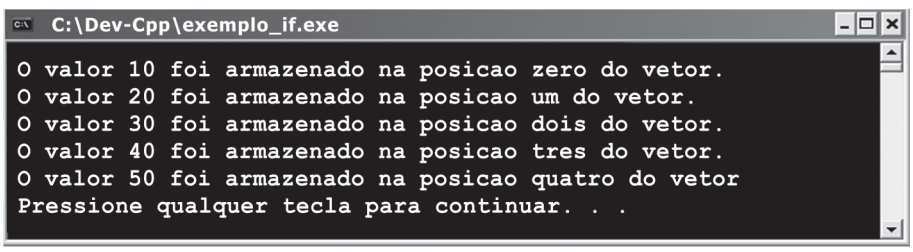


Figura 30 – Resultado da execução do programa da figura 29

Note que a única diferença entre a atribuição de valores quando utilizamos variáveis e quando utilizamos vetores é que no uso de vetores precisamos indicar o índice a ser utilizado.

Para utilizar vetores para armazenar valores obtidos através do comando *scanf*, a única diferença é também a informação do *índice* do vetor a ser utilizado. Assim, por exemplo, caso queiramos ler um número e armazená-lo no índice zero do nosso vetor, o comando ficaria assim: *scanf ("%d", &numeros[0]);*

Agora, suponha que queiramos fazer um programa para solicitar a digitação das notas de 10 alunos. Será que teríamos de escrever as dez atribuições uma a uma? Imagine então um caso onde fossem 500 valores... O exemplo 2 mostra como utilizar um comando de repetição para não ter de escrever todas essas atribuições:

Exemplo 2:

```
Linha 1  ... #include <stdio.h>
Linha 2  ... #include <stdlib.h>
Linha 3  ... int main (  )
Linha 4  ... {
Linha 5  ...      float notas[10];
Linha 6  ...      int indice;
Linha 7  ...      printf ("Lendo as notas:\n");
Linha 8  ...      for (indice=0; indice<10; indice++){
Linha 9  ...           printf ("Digite a nota do proximo aluno:");
Linha 10 ...           scanf ("%f", &notas[indice]);
Linha 11 ...      }
Linha 12 ...      printf ("Exibindo as notas digitadas:\n");
Linha 13 ...      for (indice=0; indice<10; indice++){
Linha 14 ...           printf ("A nota %f foi armazenada na posicao
                       %d do vetor.\n",notas[indice], indice);
Linha 15 ...      }
Linha 16 ...      system ("pause");
Linha 17 ... }
```

Vamos entender algumas linhas do código acima:

Linha 5 ... float notas[10];

O vetor ***notas*** foi declarado com capacidade para armazenar dez valores do tipo float.

Linha 6 ... int indice;

Foi declarada a variável ***indice*** do tipo inteira para armazenar o índice do vetor a ser utilizado.

Linha 8 ... for (indice=0; indice<10; indice++){

Será utilizado um comando *for* para variar o valor da variável *indice* de zero até nove, de forma que, cada vez que o laço for executado, uma posição diferente do vetor será utilizada.

Linha 10 ... scanf ("%f", &notas[indice]);

A nota digitada será aramazenada no vetor de notas na posição indicada pela variável *indice.*

Linha 13 ... for (indice=0; indice<10; indice++){

Um novo laço for será utilizado, agora para imprimir o conteúdo do vetor.

Linha 14 ... printf ("A nota %f foi armazenada na posicao %d do vetor.\n",notas[indice], indice);

Será impressa na tela a nota armazenada em determinada posição, bem como o índice dessa posição.

A Figura 31 exibe o resultado de uma execução desse programa.

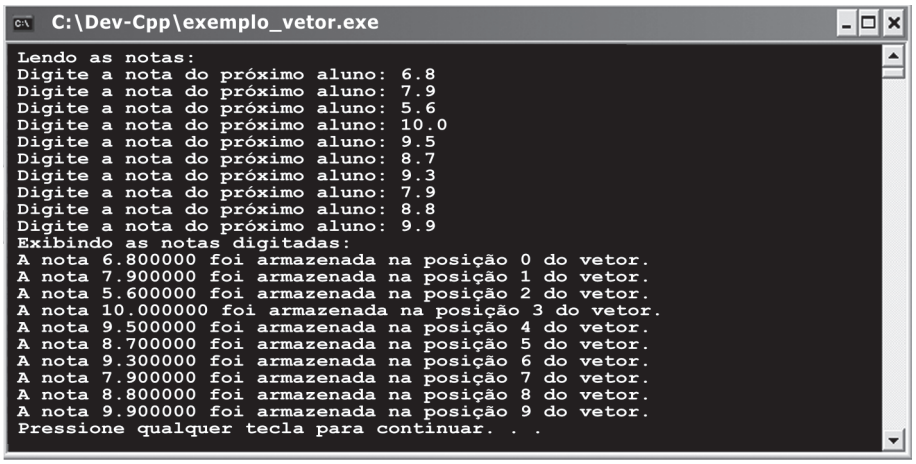


Figura 31 – Resultado da Execução do Exemplo 2

- Note que o fato de termos declarado o vetor com 10 posições não significa que estejamos livres do controle do índice.
- A linguagem C não verifica se o índice que você usou está dentro dos limites válidos. Você é quem deverá ter o cuidado de controlar os limites, como fizemos nos laços do exemplo anterior!
- O primeiro índice de um vetor é o índice zero. Assim, se tenho um vetor com "n" posições, seus índices vão de zero a "n – 1".

60. Faça um programa que:

a) preencha dois vetores A e B de 5 posições, com números inteiros;
b) atribua a um vetor RES a soma do vetor A com B (a primeira posição de A será somada à primeira posição de B e o resultado será atribuído à primeira posição do vetor RES);
c) mostre os valores do vetor RES.

61. Faça um programa que solicite a digitação e armazene 20 números reais em um vetor. Depois o programa deve ficar disponível para o usuário digitar o valor do índice para que seja exibido o número armazenado no índice solicitado. Para encerrar o programa, o usuário deve informar um índice inválido (lembre-se de que para um vetor de 20 posições os índices válidos são de 0 a 19).

62. Faça um programa que solicite a digitação de 10 números inteiros e os armazene em um vetor. Depois o programa deve ler o vetor e imprimir na tela uma listagem dos múltiplos de 2, uma outra dos múltiplos de 3 e uma última listagem dos múltiplos de 5.

6.2. *STRINGS* – VETORES DE CARACTERES

Quando estudamos Portugol, no capítulo 2, vimos que um dos tipos de variáveis existentes era o tipo *cadeia de caracteres*. Esse tipo de variável era utilizado para armazenar textos como palavras e nomes. Como vimos até agora, não há um tipo primitivo de variáveis em C que seja equivalente ao tipo *cadeia de caracteres*. Assim, como armazenaremos textos em linguagem C?

A resposta está na utilização de vetores. Para trabalhar com cadeias de caracteres em C, devemos utilizar vetores de caracteres, ou seja, vetores do tipo *char*.

Em textos técnicos de programação, muito comumente, cadeias de caracteres também são conhecidas como *string*. Uma *string* em C é um vetor do tipo *char* terminado pelo caractere nulo (‘/0’). Este detalhe “toda *string* é finalizada pelo caractere nulo” é importante, pois, se desejarmos armazenar uma cadeia de 8 caracteres, deveremos declarar um vetor de, pelo menos, 9 posições, pois a última posição conterá o caractere nulo.

Veja na figura 32 uma demonstração de um vetor de 9 posições, para armazenar a string “Educação”.

NOME								
E	D	U	C	A	C	A	O	\0
0	1	2	3	4	5	6	7	8

Figura 32 - Demonstração do vetor NOME

A declaração do nosso vetor ficará assim: **char** NOME[9];

Observe que declaramos uma posição a mais, garantindo uma posição para o caractere nulo.

6.2.1. Leitura de Strings

Para ler uma ***string*** digitada por um usuário, devemos utilizar a função ***gets*** ( ). Essa função colocará o terminador nulo na ***string***, assim que a tecla enter for pressionada.

Vejamos um exemplo de leitura de uma cadeia de caracteres utilizando ***gets*** ( ).

Exemplo 3:

```
Linha 1  ... #include <stdio.h>
Linha 2  ... #include <stdlib.h>
Linha 3  ... int main ( )
Linha 4  ... {
Linha 5  ...      char nome[20];
Linha 6  ...      printf (“Digite seu nome: “);
Linha 7  ...      gets (nome);
Linha 8  ...      printf (“O nome digitado foi: %s \n”,nome);
Linha 9  ...      system (“pause”);
Linha 10 ... }
```

Vamos entender algumas linhas do código:

linha 5 ... char nome[20];

Declaramos um vetor com 20 posições. Isso significa que esse vetor poderá armazenar até 19 caracteres, pois um caractere será utilizado para armazenar o caractere nulo que indica o fim da string.

linha 7 ... gets (nome);

Observe que para leitura não utilizamos a função scanf ( ); a função utilizada para leitura da string foi a gets ( ).

linha 8 ... printf (“O nome digitado foi %s \n”,nome);

O que nos chama a atenção nessa linha é o caractere de impressão %s. Esse caractere, como já vimos, imprime uma cadeia de caracteres.

A Figura 33 exibe o resultado da execução desse programa.

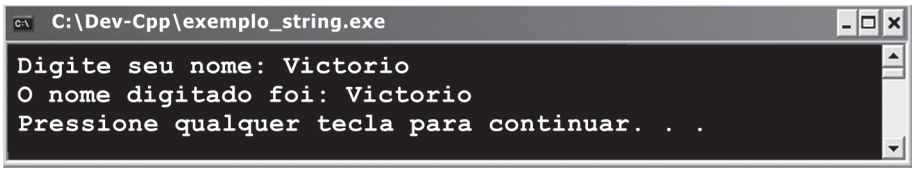


Figura 33 – Resultado da Execução do Exemplo 3

O que aprendemos até aqui?

- Vetor é uma estrutura indexada que armazenará dados de um mesmo tipo (homogêneos).
- Uma string é, na realidade, um vetor de caracteres e deverá ser lida usando a função ***gets*** ( ).

.

6.3. MATRIZES

Matrizes são estruturas indexadas utilizadas para representar certa quantidade de valores de um mesmo tipo.

Se você estiver atento, deve estar se perguntando: “Mas esse não é o conceito de vetor?”.

Na verdade, vetor é um tipo especial de matriz. Dizemos que vetor é uma matriz unidimensional. Ou seja, enquanto o vetor tem apenas uma dimensão, a matriz pode ter tantas dimensões quantas forem necessárias. Mas, neste curso trabalharemos sempre com matrizes de duas dimensões. Assim, neste texto, sempre que nos referirmos a matrizes, estaremos falando de matrizes bidimensionais.

Para facilitar o entendimento, a Figura 34 apresenta uma matriz capaz de armazenar duas notas de uma turma de cinco alunos.

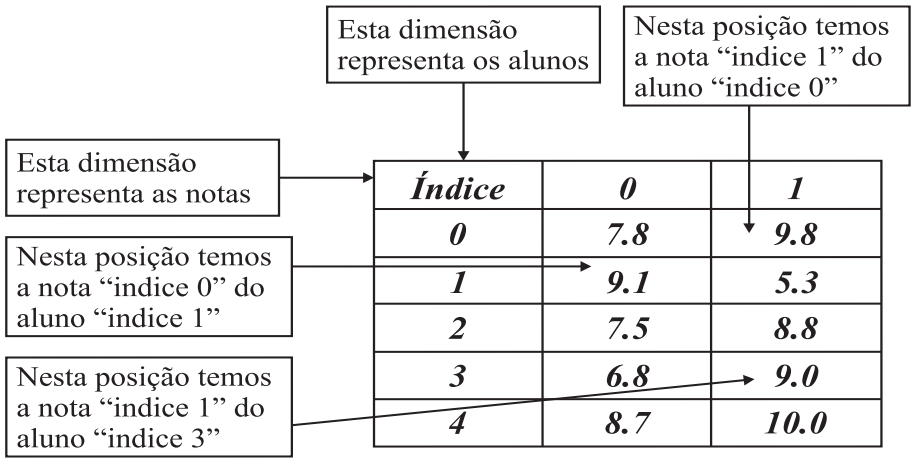


Figura 34 – Ilustração de uma Matriz

Suponha que o primeiro aluno da lista de chamada de uma turma seja o André e o segundo seja o Bruno. Assim, suponha que a matriz da Figura 34 esteja armazenando as notas dessa turma. A nota do André (indice 0 na dimensão de alunos) na primeira prova (indice 0 na dimensão de notas) foi 7.8. Já a nota do Bruno (indice 1 na dimensão de alunos) na segunda prova (indice 1 na dimensão de notas) foi 5.3.

Sintaxe para declaração de matrizes de duas dimensões:
tipo_da_matriz nome_da_matriz[altura][largura];

Como funciona:

A única diferença entre a declaração de matrizes de duas dimensões e a declaração de vetores é que, ao declarar vetores, informamos apenas uma dimensão (o seu tamanho), enquanto em matrizes bidimensionais informamos duas dimensões.

Vale ressaltar que a linguagem C permite a criação de matrizes de quantas dimensões o programador quiser, bastando acrescentar as dimensões entre colchetes [ ].

Assim, uma matriz de duas dimensões é como um vetor onde cada elemento é também um vetor.

A declaração da matriz apresentada na Figura 34 ficará assim: float notas [5][2];

Na declaração informamos que:

- **float** - armazenará apenas valores do tipo float;
- **notas [5][2]** – a matriz será acessada através do nome "notas". O primeiro índice "[5]" indexa as linhas e o segundo índice da direita "[2]" indexa as colunas.

A seguir temos um código de exemplo que alimenta a matriz da *Figura 34* e depois a exibe:

Exemplo 4:

```
Linha 1   ... #include <stdio.h>
Linha 2   ... #include <stdlib.h>
Linha 3   ... int main (   )
Linha 4   ... {
Linha 5   ...     float notas[5][2];
Linha 6   ...     int lin,col;
Linha 7   ...     printf ("INICIANDO O LOOP DE LEITURA \n \n");
Linha 8   ...     for (lin=0;lin<5;lin++)
Linha 9   ...         for (col=0;col<2;col++) {
Linha 10  ...             printf ("Digite a nota %d do aluno %d: ",
                          col, lin);
Linha 11  ...             scanf ("%f", &notas[lin][col]);
```

```
Linha 12 ...      }
Linha 13 ... printf (“\nINICIANDO O LOOP DE EXIBICAO \n \n”);
Linha 14 ... for (lin=0;lin<5;lin++)
Linha 15 ...      for (col=0;col<2;col++)
Linha 16 ...            printf (“nota = %.2f \n”,notas[lin][col]);
Linha 17 ... system (“pause”); }
```

Nesse código vale destacar as seguintes linhas:

linha 8... for (lin=0;lin<5;lin++)

linha 9... for (col=0;col<2;col++)

O laço ***for*** da linha 8 controla o acesso às linhas. Dentro desse ***for*** há um outro ***for***, na linha 9, que é responsável pelo controle das colunas. Assim, o preenchimento da matriz se dá linha a linha, ou seja, só após preencher todas as colunas da linha zero é que se começa a preencher a linha 1 e assim sucessivamente.

Note que há um controle análogo para a impressão da matriz nas linhas 14 e 15.

A figura 35 exibe o resultado da execução do exemplo 4, entrando com os mesmos dados apresentados na figura 34.

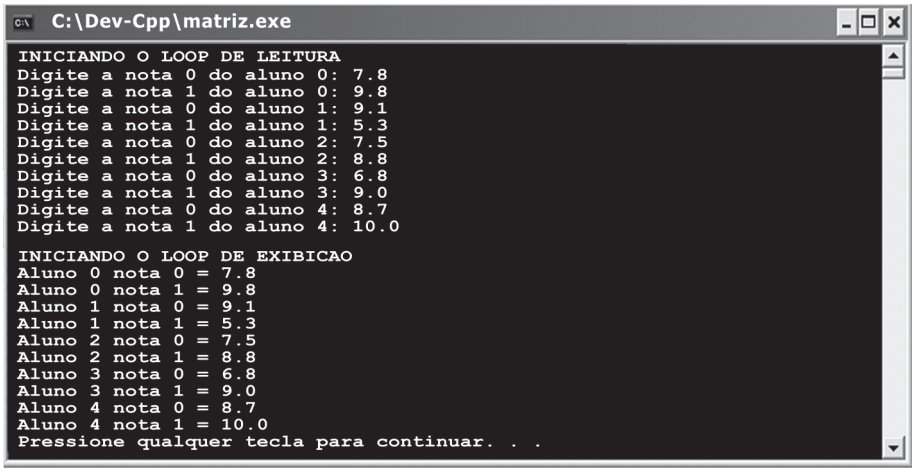


Figura 35 – Resultado da execução do exemplo 4

Vale a pena lembrar:

- Na linguagem C, os índices iniciam de zero;
- É responsabilidade do programador manter o controle sobre a faixa de índices acessíveis. A Linguagem C não fará o controle automaticamente. Ou seja, se você definir um vetor de 10 posições (índices de 0 a 9) e tentar acessar a posição de índice 10, não ocorrerão erros de compilação. Mas, ao executar o programa, será acessada uma posição de memória não alocada para o mesmo e estará sujeito a erros.

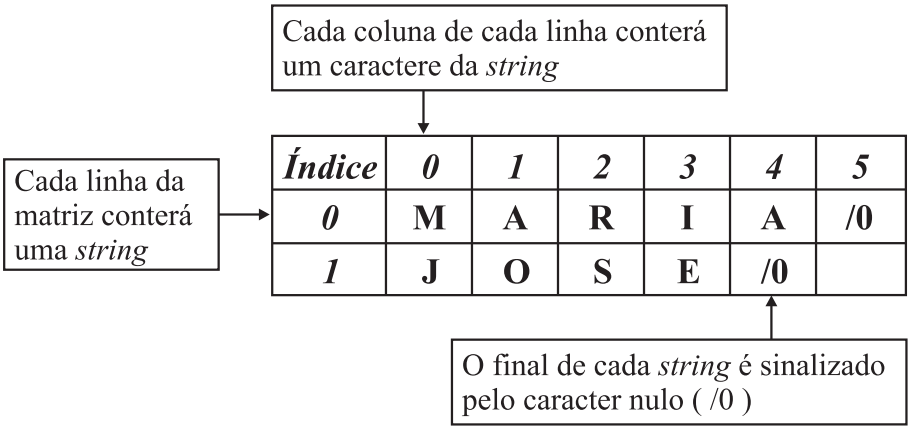
63. Utilizando matriz, faça um programa que leia duas notas de dez alunos e depois calcule e armazene a média dos mesmos. Você deve utilizar uma matriz de 10 linhas por 3 colunas. Cada linha representará as notas de um aluno. Na primeira coluna serão armazenadas as notas da primeira avaliação, na segunda coluna as notas da segunda avaliação e na terceira coluna a média calculada. Para finalizar, exiba a quantidade de alunos com média abaixo de 6.0, a quantidade acima de 6.0 e calcule a média da turma fazendo a média aritmética entre as médias dos alunos.

64. Carregue uma matriz 3 x 3 com os valores das vendas de uma loja, supondo 3 meses e 3 vendedores. Calcule e mostre, em cada mês, qual foi o vendedor que vendeu mais.

6.3.1. Matrizes de Caracteres

Como estudamos na seção 6.2, em linguagem C as *strings* são representadas por meio de vetores de caracteres. Assim, caso precisemos de uma estrutura de dados que armazene várias strings, teremos de utilizar matrizes de caracteres.

Para facilitar o entendimento, a figura 36 exibe uma matriz capaz de armazenar duas strings de cinco caracteres:



Índice	*0*	*1*	*2*	*3*	*4*	*5*
0	M	A	R	I	A	/0
1	J	O	S	E	/0	

Figura 36 – Matriz de Strings

No exemplo 5 apresenta-se um programa que utiliza uma matriz de caracteres para armazenar o nome de três pessoas, sendo que cada nome poderá ter, no máximo, 29 caracteres.

Exemplo 5:

```
Linha 1   ... #include <stdio.h>
Linha 2   ... #include <stdlib.h>
Linha 3   ... int main (   )
Linha 4   ... {
Linha 5   ... char nomes[3][30];
Linha 6   ... int lin;
Linha 7   ...printf ("INICIANDO O LOOP DE LEITURA \n \n");
Linha 8   ... for (lin=0;lin<3;lin++)
Linha 9   ... {
Linha 10 ...          printf ("Digite o nome: ");
Linha 11 ...          gets (nomes[lin]);
Linha 12 ... }
Linha 13 ...    printf ("\nINICIANDO O LOOP DE EXIBICAO \n
                \n");
Linha 14 ... for (lin=0;lin<3;lin++)
Linha 15 ...          printf ("nome = %s \n",nomes[lin]);
Linha 16 ... system ("pause");
Linha 17 ... }
```

Vamos entender algumas linhas do código:

linha 5 ... char nomes[3][30];

Declaramos a matriz para receber 3 nomes de, no máximo, 30 caracteres. ***Lembre-se de que uma posição da string deve ser reservada para o caractere nulo. Por isso, apesar de a segunda dimensão de nossa matriz ser 30, só poderemos armazenar nomes de, no máximo, 29 caracteres***.

linha 6 ... int lin;

Declaramos a variável lin que será utilizada como índice para nossa matriz.

linha 7 ... printf ("INICIANDO O LOOP DE LEITURA \n \n");

O printf imprimirá uma mensagem avisando o início da leitura.

linha 8 ... for (lin=0;lin<3;lin++)

Como a matriz é de strings, só precisamos de um **for** para controlar o índice linha.

linha 11 ... gets (nomes[lin]);

Utilizamos a função gets ( ) para ler os nomes (os caracteres). **Note que utilizamos apenas o índice de linha, pois cada linha conterá uma *string*. Cada coluna de cada linha conterá um caractere da *string*.**

linha 15 ... for (lin=0;lin<3;lin++)

Exibindo os nomes digitados.

A figura 37 exibe o resultado de uma execução do exemplo 5.

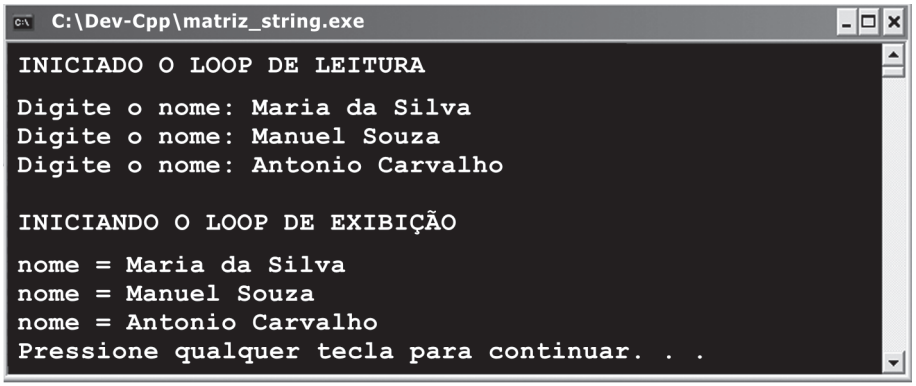
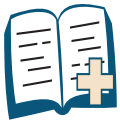


Figura 37 – Resultado da Execução do exemplo 5

65. Crie um programa que leia o nome e o salário de 20 pessoas. Os nomes devem ser armazenados em uma matriz de caracteres. Os nomes devem ter, no máximo, 15 caracteres. Assim, a matriz deve ser capaz de comportar 20 nomes de 15 caracteres. Os salários devem ficar armazenados em um vetor de números reais. Assim, o nome armazenado na posição x da matriz terá seu salário armazenado na posição x do vetor. Após ler todos os 20 nomes e salários, o programa deve ficar disponível para o usuário digitar o valor do índice para que seja exibido o nome do funcionário armazenado nesse índice e seu respectivo salário. Para encerrar o programa, o usuário deve informar um índice inválido (lembre-se de que, para um vetor de 20 posições, os índices válidos são de 0 a 19).

66. Crie um programa que leia e armazene o nome (no máximo 10 caracteres), o valor de compra e o valor de venda de 10 produtos. Os valores de compra e venda devem ser armazenados em uma única matriz de números reais, enquanto os nomes devem ficar em uma matriz de caracteres. Após a leitura, o programa deve ficar disponível para o usuário digitar o valor do índice para que seja exibido o nome do produto armazenado nesse índice e o lucro obtido com a venda desse produto. O lucro é dado pela diferença entre o valor de venda e o valor de compra. Para encerrar o programa, o usuário deve informar um índice inválido.



Leituras complementares:

SCHILDT, Herbert. **C Completo e Total**. São Paulo: Pearson, 2006.

KERNIGHAN Brian W. **C Linguagem de Programação Padrão ANSI**. Rio de Janeiro: Elsevier, 1989.

7. FUNÇÕES E PROCEDIMENTOS

Caro aluno,

Estamos chegando à reta final da nossa disciplina! Neste último capítulo, vamos conhecer um conceito fundamental da programação: o conceito de função.

Assim como nos capítulos anteriores, é fundamental que você entenda os conceitos e os coloque em prática.

Esta disciplina é apenas a primeira etapa da viagem pelo mundo da programação que faremos durante este curso. Assim, o entendimento dos assuntos nela abordados é fundamental para a sequência do curso.

7.1. MODULARIZAÇÃO

Através das experiências nos capítulos anteriores, você deve ter notado que quanto maior e mais complexo é o programa, mais difícil é para se compreender seu funcionamento. Assim, para tentar reduzir essa complexidade e facilitar o desenvolvimento e a manutenção dos programas, surge a ideia de modularização.

A ideia da modularização baseia-se no famoso princípio de "dividir para conquistar". As técnicas de modularização têm como objetivo principal dividir um programa grande e complexo em vários programas menores e mais simples. Assim, são desenvolvidos vários "programas" menores que se integram para resolver o problema maior.

Como exemplo, suponha que tenhamos de desenvolver um programa para calcular a folha de pagamento de uma empresa. Esse problema grande e complexo pode ser divido em partes menores, como:

- OBTER OS DADOS DOS FUNCIONÁRIOS
- CALCULAR INSS
- CALCULAR FGTS
- CALCULAR IMPOSTO DE RENDA
- CALCULAR FÉRIAS
- CALCULAR SALÁRIO LÍQUIDO
- IMPRIMIR OS CONTRACHEQUES

Note que conseguimos dividir nosso problema inicial em vários problemas menores, ou seja, começamos a modularizar nosso programa.

A maneira mais simples de modularizar nossos programas é através da sua divisão em funções ou em procedimentos. Voltando ao nosso exemplo, cada uma das partes do nosso problema seria resolvida através da construção de uma função ou de um procedimento.

Dentre os benefícios obtidos com a modularização dos programas, podemos destacar:

- ***Maior facilidade no desenvolvimento do programa:*** é muito mais fácil pensar em cada um dos programas menores do que no programa maior como um todo;
- ***Maior facilidade para testar os programas****:* é mais fácil verificar o funcionamento e encontrar as falhas em pequenos programas com objetivos específicos do que em um grande e complexo programa;
- ***Maior facilidade na leitura do programa (legibilidade):*** programas modularizados são mais fáceis de serem lidos e entendidos;
- ***Maior facilidade para efetuar manutenção (manutenibilidade):*** é mais fácil alterar programas modularizados. Por exemplo, imagine se a tabela de recolhimento do imposto de renda for alterada. Se nosso programa estiver modularizado como pensamos acima, basta alterar a função que calcula o imposto de renda e nada mais será afetado.
- ***Maior facilidade de reutilização****:* a modularização facilita o reaproveitamento de trabalhos executados anteriormente. Por exemplo, suponha que tenhamos construído nosso programa de folha de pagamento para uma certa empresa. Se precisarmos fazer um programa de folha de pagamento para uma outra empresa, certamente teremos de alterar as funções de obter dados e de imprimir os contracheques. Talvez precisemos alterar algo na função que calcula as férias. Mas as demais funções provavelmente poderão ser reutilizadas sem alterações.

7.2. FUNÇÕES E PROCEDIMENTOS

Uma função é uma unidade de código de programa autônoma desenvolvida para cumprir uma determinada tarefa em particular (MIZRAHI, 1990).

A única diferença que há entre os conceitos clássicos de procedimento e função está no fato de que uma função sempre retorna um valor como resultado de sua execução, enquanto um procedimento não provê um retorno.

A linguagem C fornece ao programador um conjunto de funções e procedimentos prontos para o uso. Nós já utilizamos algumas delas, como a *printf ( )* e a *scanf ( )*.

Mas, como poderíamos criar nossas próprias funções e procedimentos em C?

7.2.1. Criando funções em Linguagem C

Para criar nossas funções em linguagem C, precisamos seguir uma sintaxe padrão. Essa sintaxe é exibida no quadro abaixo:

Sintaxe:

tipo_de_retorno nome_da_função
(declaração_de_parâmetros)
{
corpo_da_função
}

Onde:

tipo_de_retorno: é o tipo de dado a ser retornado pela função. Os tipos de retornos possíveis são exatamente os mesmos tipos de variáveis que já vimos.

nome_da_função: assim como demos nomes a nossas variáveis, precisamos nomear nossas funções. É através desse nome que vamos executar a função. É importante dar nomes que reflitam o objetivo da função.

declaração_de_parâmetros: uma função pode receber dados como entrada para efetuar suas ações. Esses dados são conhecidos como **parâmetros**. A declaração de parâmetros funciona de forma parecida com a declaração de variáveis.

corpo_da_função: é o conjunto de comandos que a função irá executar.

Para exemplificar, vamos utilizar nosso exemplo da folha de pagamento. Para simplificar, vamos descontar do salário bruto apenas o INSS. Assim, vamos fazer uma função que calcule o desconto do INSS e utilizar essa função em nosso programa.

Exemplo 1:

```
Linha 1   ... #include <stdio.h>
Linha 2   ... #include <stdlib.h>
Linha 3   ... /*DEFINICAO DA FUNCAO*/
Linha 4   ... float Calcular_INSS (float salario){
Linha 5   ...    float desconto_inss;
Linha 6   ...    desconto_inss= salario*0.08;
Linha 7   ...    return (desconto_inss);
Linha 8   ... }
Linha 9   ... //INICIANDO O PROGRAMA PRINCIPAL
```

```
Linha 10 ... int main (   ){
Linha 11 ...    float salario_bruto, salario_liquido, inss;
Linha 12 ...    printf (“Digite o salario do fucionario: “);
Linha 13 ...    scanf (“%f”,&salario_bruto);
Linha 14 ...    inss = Calcular_INSS (salario_bruto);
Linha 15 ...    salario_liquido = salario_bruto - inss;
Linha 16 ...    printf (“Salario liquido = %f \n\n “,salario_liquido);
Linha 17 ...    system (“PAUSE”);
Linha 18 ...    return 0;
Linha 19 ... }
```

Vamos entender algumas linhas do nosso exemplo:

linha 3 ... /*DEFINICAO DA FUNCAO*/

Comentário apenas indicando que estamos iniciando a função. Lembre-se de que, em linguagem C, tudo o que está entre /* e */ é comentário, serve apenas para facilitar o entendimento do código. Vale lembrar também que, na linguagem C, há o comentário de uma linha, que é definido com o uso de //.

linha 4 ... float Calcular_INSS (float salario){

Essa linha define a nossa função. Note que:

- O nome da função é Calcular_INSS;
- Nossa função irá retornar um valor do tipo *float*;
- Essa função receberá como parâmetro um valor do tipo float. Esse parâmetro será chamado de *salario.*

linha 5 ... float desconto_inss;

Declaramos uma variável chamada desconto_inss que utilizaremos dentro da função. ***Vale ressaltar que essa variável só pode ser utilizada dentro dessa função.***

linha 6 ... desconto_inss = salario*0.08;

Calculamos o valor do desconto do INSS e armazenamos esse valor na variável *desconto_inss.* Note que, para calcular o valor do desconto, utilizamos o salário que foi passado como parâmetro para a função.

linha 7 ... return (desconto_inss);

Como explicamos, toda função deve retornar um valor. Em linguagem C, esse retorno é feito através do uso da função *return ( ).* Assim, nesta linha, estamos retornando o valor do desconto do inss para a função que executar a nossa função. É importante notar que quando declaramos

nossa função, **na linha 4**, definimos que ela retornaria um valor do tipo *float,* que é exatamente o tipo da variável *desconto_inss* que está sendo retornada. Essa compatibilidade de tipos é obrigatória. Não poderia, por exemplo, definir que o tipo de retorno de uma função é *float* e retornar através do comando *return ( )* um valor de outro tipo.

linha 10 ... int main ( ){

Aqui estamos iniciando nossa função principal. Lembre-se de que, em linguagem C, é obrigatória a definição da função main, que é a primeira função do programa a ser executada.

linha 11 ... float salario_bruto, salario_liquido, inss;

Declaração das variáveis que serão utilizadas no programa principal. ***Vale ressaltar que essas variáveis só podem ser utilizadas dentro da função main.***

linha 14 ... inss = Calcular_INSS (salario_bruto);

Aqui estamos fazendo uma chamada à nossa função Calcular_INSS, passando como parâmetro a variável *salario_bruto* para a qual foi lido o valor digitado pelo usuário na **linha 13**. O retorno da execução dessa função está sendo atribuído à variavel *inss*. Assim, o valor que for retornado pela função será armazenado na variável *inss*.

linha 15 ... salario_liquido = salario_bruto – inss;

Estamos subtraindo o desconto de INSS (que foi calculado pela nossa função) do valor do salário bruto para obtermos o salário líquido.

A *figura 38* exibe o resultado da execução desse programa, entrando com um salário de R$ 1.000,00.

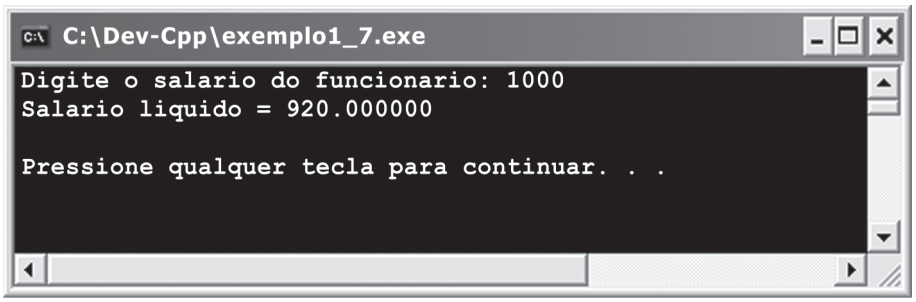


Figura 38 – Exibição do Resultado da Execução do Exemplo 1 tendo como entrada um salário de R$ 1.000,00

Mas, como é o fluxo de execução do programa exemplo?

Como sabemos, um programa em linguagem C começa a ser executado pela função *main ( )*. Assim, a execução do programa exemplo começaria pela *linha 10*.

Na *linha 11* são declaradas as variáveis e na *13* é lido o valor do salário bruto. Então, na *linha 14* é feita uma chamada à função *Calcular_INSS ( )*. Assim, a execução é transferida para a *linha 4*.

Na *linha 6* o valor do desconto de inss é calculado e armazenado na variável *desconto_inss*. Na *linha 7* utilizamos o comando *return ( )* para retornar o valor da variável *desconto_inss* para o ponto onde a função foi chamada. Assim, voltamos à *linha 14* atribuindo à variável *inss* o valor de retorno da função chamada. Então, as linhas seguintes são executadas sequencialmente até o fim do programa.

7.2.2. Criando Procedimentos em Linguagem C

Como citado no início do capítulo, a diferença entre os conceitos de função e procedimento está no fato de "uma função sempre gerar um retorno", enquanto "um procedimento não gera retorno". Em linguagem C não há uma sintaxe para definição de procedimento. Quando uma função em C não vai retornar valor algum, definimos o tipo de retorno dessa função *void*. Assim, em linguagem C, definimos procedimentos como funções que têm como tipo de retorno o tipo especial *void*.

Para facilitar o entendimento, vamos refazer o *exemplo 1* mas, agora, ao invés de apenas imprimir o salário líquido utilizando um printf, vamos definir um procedimento que receberá o salário bruto, o líquido e o desconto de INSS e simplesmente imprimirá as informações na tela.

Exemplo 2:

```
Linha 1  ... #include <stdio.h>
Linha 2  ... #include <stdlib.h>
Linha 3  ... /*Definicao da funcao que calcula o INSS*/
Linha 4  ... float Calcular_INSS (float salario){
Linha 5  ...     float desconto_inss;
Linha 6  ...     desconto_inss= salario*0.08;
Linha 7  ...     return (desconto_inss);
Linha 8  ... }
Linha 9  ... //Definicao da funcao que imprime os dados na tela
Linha 10 ... void Imprime_Dados (float bruto, float liquido, float
                 inss){
Linha 11 ...     printf ("Seu salario bruto e: %.2f\n", bruto);
Linha 12 ...     printf ("Sua contribuicao para o INSS e: %.2f\n", inss);
Linha 13 ...     printf ("Assim, seu salario liquido e: %.2f\n", liquido);
Linha 14 ... }
Linha 15 ... //INICIANDO O PROGRAMA PRINCIPAL
Linha 16 ... int main ( ){
Linha 17 ...     float salario_bruto, salario_liquido, inss;
```

```
Linha 18 ...    printf (“Digite o salario do fucionario: “);
Linha 19 ...    scanf (“%f”,&salario_bruto);
Linha 20 ...    inss = Calcular_INSS (salario_bruto);
Linha 21 ...    salario_liquido = salario_bruto - inss;
Linha 22 ...    Imprime_Dados (salario_bruto, salario_liquido, inss);
Linha 23 ...    system (“PAUSE”);
Linha 24 ...    return 0;
Linha 25 ... }
```

No *exemplo 2*, da linha 10 à linha 14 definimos uma função para exibir os dados na tela. Vamos entender algumas linhas do nosso código:

linha 10•void Imprime_Dados (float bruto, float liquido, float inss){

Essa linha define o nosso procedimento que irá imprimir os dados na tela. Note que:

- O nome do procedimento é Imprime_Dados;
- Nosso procedimento não está retornando dado algum (não utilizamos nenhum *return ( )* nela). Assim, o tipo de retorno foi definido como *void*;
- O procedimento está recebendo três parâmetros, todos do tipo float.

linha 11 ... printf (“Seu salario bruto e: %.2f\n”, bruto);

linha 12... printf (“Sua contribuicao para o INSS e: %.2f\n”, inss);

linha 13... printf (“Assim, seu salario liquido e: %.2f\n”, liquido);

Essas três linhas imprimem, nessa ordem, os valores do salário bruto, da contribuição para o INSS e do salário líquido. Ao terminar o procedimento, a execução volta à primeira linha do programa após a chamada do procedimento.

linha 22.. Imprime_Dados (salario_bruto, salario_liquido, inss);

Aqui está sendo feita a chamada ao procedimento *Imprime_Dados*, passando como parâmetros as variáveis *salario_bruto, salario_liquido, inss*. É importante notar a ordem de passagem dos parâmetros. Qualquer mudança nessa ordem ocasionará erros. Da forma como foi feita a chamada, o valor da variável *salario_bruto* está sendo passado para o parâmetro *bruto,* o valor da variável *salario_liquido* está sendo passado para o parâmetro *liquido* e o valor da variável *inss* está sendo passado para o parâmetro *inss.*

É importante notar também que precisamos passar essas variáveis como parâmetros, pois, como elas foram definidas dentro da função *main ( )*, elas só existem no contexto daquela função.

A figura 39 exibe o resultado da execução do *exemplo 2*, entrando com um salário de R$ 1.000,00.

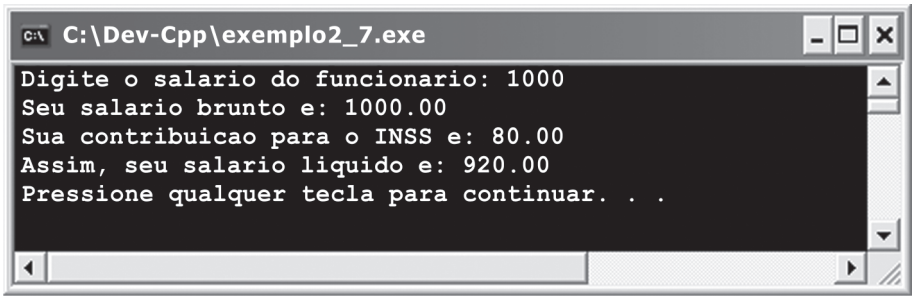


Figura 39 – Exibição do resultado da execução do exemplo 2 tendo como entrada um salário de R$ 1.000,00

O que aprendemos até aqui?

- Modularização é uma técnica de programação que utilizamos para dividir um programa maior em programas menores. Aos programas menores denominamos funções ou procedimentos.
- A diferença entre função e procedimento está no fato de funções fornecerem um retorno, enquanto procedimentos não o fazem.
- Procedimentos são definidos em linguagem C como funções cujo retorno é do tipo *void*.
- Variáveis definidas dentro de uma função só podem ser utilizadas dentro da própria função, pois, ao finalizar a execução da função, elas são "apagadas" da memória.

7.3. PASSAGEM DE PARÂMETROS

Como vimos nos exemplos 1 e 2, parâmetros são os valores que passamos como entrada para as funções e procedimentos. Há duas formas de se fazer passagem de parâmetros: por valor ou por referência.

7.3.1. Passagem de parâmetros por valor

Quando passamos um parâmetro por valor para uma função, mesmo que a variável passada como parâmetro sofra alterações dentro da função chamada, o valor dela não é alterado no contexto do programa principal. Todas as passagens de parâmetros que fizemos até aqui em nossos exemplos foram passagens por valor.

Para facilitar o entendimento, vamos ao exemplo 3:

```
#include <stdio.h>
#include <stdlib.h>
void Imprime (int x) {
    //Altero o valor de x somando 1 ao valor passado
    x++;
    printf("Valor do parametro dentro da função: %d \n", x);
}
int main() {
    int x;
    printf("Digite um valor inteiro: ");
    scanf("%d", &x);
    Imprime (x);
    printf("Valor da variavel dentro do main: %d \n", x);
    system("pause");
}
```

Figura 40 – Exemplo 3

A **Figura 41** exibe o resultado da execução do exemplo 3, fornecendo como entrada o valor 1.

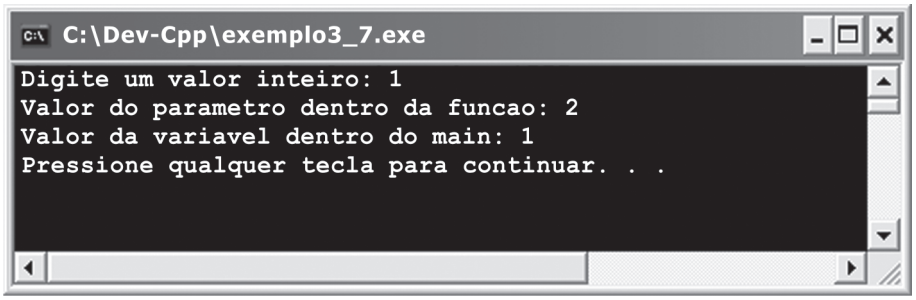


Figura 41 – Resultado da Execução do Exemplo 3

Note que chamamos a função *Imprime ( )* passando como parâmetro a variável *x* que tinha valor =1. Dentro da função incrementamos o argumento da função e imprimimos seu valor para mostrar que agora ela valia 2. Quando a execução da função finalizou e voltamos à função *main,* imprimimos novamente o valor da variável *x* e vimos que, mesmo com o incremento do parâmetro dentro da função, a variável continua com o valor antigo. Ou seja, a alteração de valor feita dentro da função *Imprime ( )* não se refletiu fora dessa função.

7.3.2. Passagem de Parâmetros por Referência

Quando queremos que as alterações nos valores de um determinado parâmetro sejam refletidas no valor da variável passada como argumento, devemos utilizar a passagem de parâmetro por referência.

Em linguagem C, para simular a passagem de parâmetro por referência, utilizamos ponteiros. Os ponteiros são variáveis utilizadas para fazer referência a outras variáveis. Neste momento não devemos nos preocupar com o conceito de ponteiro. O importante é entender como fazer a passagem por referência.

Para indicar a passagem por referência, na definição da função devemos colocar um asterisco (*) na frente do nome do parâmetro e na chamada da função devemos colocar um & na frente da variável. Um exemplo de passagem por referência que utilizamos sem nos dar conta é na função *scanf ( )*. Está lembrado que, quando passamos uma variável para o *scanf ( )* colocamos um & na frente do nome da variável? É exatamente por isso que a função *scanf ( )* consegue alterar o valor das variáveis.

Para facilitar o entendimento, vamos ao exemplo 4:

```
Linha 1  ... #include <stdio.h>
Linha 2  ... #include <stdlib.h>
Linha 3  ... void Imprime (int *x){
Linha 4  ...    (*x)++;
Linha 5  ...    printf ("Valor do parametro dentro da funcao:
                %d\n", *x);
Linha 6  ... }
Linha 7  ... int main (  ){
Linha 8  ...    int x;
Linha 9  ...    printf ("Digite um valor inteiro:");
Linha 10 ...    scanf ("%d",&x);
Linha 11 ...    Imprime (&x);
Linha 12 ...    printf ("Valor da variavel dentro do main: %d\n", x);
Linha 13 ...    system ("pause");
Linha 14 ... }
```

linha 3 ... void Imprime (int *x){

Note que, na definição da função, para indicar que x será um parâmetro passado por referência, colocamos um asterisco (*) na frente do nome do parâmetro.

linha 4 ... (*x)++;

linha 5 ... printf ("Valor do parametro dentro da funcao: %d\n", *x);

Dentro da função imprimir, como o parâmetro x foi passado por referência, sempre colocamos o asterisco (*) quando o utilizamos.

linha 11 ... Imprime (&x);

Na chamada à função *Imprime ( )* colocamos um & na frente do nome da variável para indicar a passagem por referência.

A figura 42 exibe o resultado da execução do exemplo 4, fornecendo como entrada o valor 1. Note que, depois da execução do procedimento *Imprime ( )*, o valor da variável é *2* e não mais *1,* pois, como a passagem de parâmetro foi feita por referência, as alterações feitas dentro da função refletem no valor da variável fora da função.

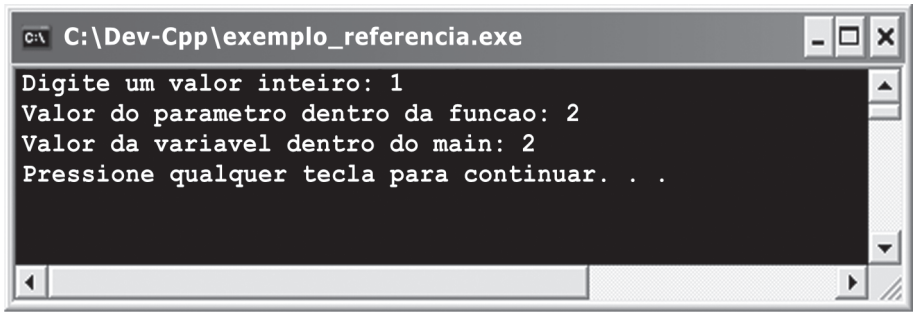


Figura 42 – Resultado da Execução do Exemplo 4

parâmetro: o termo parâmetro muitas vezes é utilizado como sinônimo de **argumento**, mas geralmente utilizamos "*parâmetros*" quando fazemos referência às variáveis situadas na assinatura de um método ou função e "*argumentos*" aos valores atribuídos a esses parâmetros. (BRIAN, 1978, p.25).

passagem de parâmetro por referência: a mudança do valor de um parâmetro dentro de uma função afeta o valor da variável original. (BRIAN, 1978, p.26).

passagem de parâmetro por valor: alterações no parâmetro não afetam a variável externa. (BRIAN, 1978, p.26).

67. Crie um programa que, usando uma função, retorne o valor da divisão de dois números inteiros para o programa principal e mostre o resultado.

    FUNÇÃO MAIN – pedirá os dois números inteiros, chamará a função DIVISAO e exibirá o resultado.

    FUNÇÃO DIVISÃO – dividirá os números e retornará o valor da divisão para main ( ).

68. Criar um programa que apresente um menu de seleções no programa principal. Esse menu deverá dar ao usuário a possibilidade de escolher uma entre quatro operações aritméticas. Escolhida a opção desejada, deverá ser solicitada a entrada de dois números. Processada a operação, o resultado deverá ser exibido. O programa deverá ser executado enquanto o usuário quiser.

    FUNÇÃO MAIN:

    - Chamará o procedimento Desenha_Menu que mostrará as opções na tela,
    - Receberá a opção do usuário,
    - Enquanto a opção do usuário for diferente de "0-sair", pedirá dois números inteiros,
    - Utilizará o comando ***switch*** para chamar a função que corresponda à opção do usuário.

    FUNÇÃO DESENHA_MENU - Mostrará um menu de opções ("0-sair"), ("1–dividir"), ("2-somar"), ("3–multip") e ("4-subtrair").

    FUNÇÃO DIVIDIR – dividirá os valores lidos e mostrará o resultado.

    FUNÇÃO SOMAR– somará os valores lidos e mostrará o resultado.

    FUNÇÃO MULTIPLICAR– multiplicará os valores lidos e mostrará o resultado.

    FUNÇÃO SUBTRAIR – subtrairá os valores lidos e mostrará o resultado.

69. Faça um programa que leia os valores dos salários atuais dos funcionários de uma empresa e imprima os valores com aumento. A função main deve pedir ao usuário para digitar o salário de um funcionário e chamar a função Calcula_Aumento ( ), receber o retorno desta função, imprimir o novo valor do salário e perguntar se o usuário deseja digitar outro salário ou não. Se o usuário responder que sim, o ciclo deve se repetir até que o usuário responda que não. A função Calcula_Aumento deve seguir a seguinte regra para calcular o novo salário: se o salário for menor ou igual a R$ 500,00, deve aplicar um aumento de 0.10 (10%) e se for maior que R$ 500,00, o aumento deve ser de 0.08 (8%). Essa função deve calcular o salário e retornar o valor para a função main ( ).

7.4. PROTÓTIPO DE FUNÇÃO

Até o momento, todas as funções que fizemos foram definidas antes da função principal (*main ( )*), pois, se não fizéssemos assim, o compilador acusaria um erro ao tentar compilar o programa porque, no momento em que fosse chamar a função, ainda não saberia o formato da mesma.

Usando *protótipo de função*, poderemos apenas declarar as funções antes da função principal e escrever o corpo das funções após o programa principal. Dessa forma, o compilador tomará conhecimento do seu formato antes da compilação. O protótipo de função consiste apenas em apresentar a *assinatura da função*, ou o seu cabeçalho: o tipo de retorno, o nome da função e os parâmetros que ela receberá.

No exemplo 5, utilizamos o mesmo programa apresentado no exemplo 4, mas utilizando protótipo de função:

```
void Imprime (int x);
int main() {
    int x;
    printf("Digite um valor inteiro: ");
    scanf("%d", &x);
    Imprime (&x);
    printf("Valor da variavel dentro do main: %d \n", x);
    system("pause");
}
void Imprime (int *x) {
    //Altero o valor de x somando 1 ao valor passado
    (*x)++;
    printf("Valor do parametro dentro da funcao: %d \n", *x);
```

Figura 43 – Exemplo 5

A Figura 44 exibe o resultado da execução do exemplo 5, fornecendo como entrada o valor 1. Note que o resultado é o mesmo apresentado pelo exemplo 4. Ou seja, a utilização do protótipo é apenas para melhorar a organização do código e não altera em nada o resultado da execução do programa.

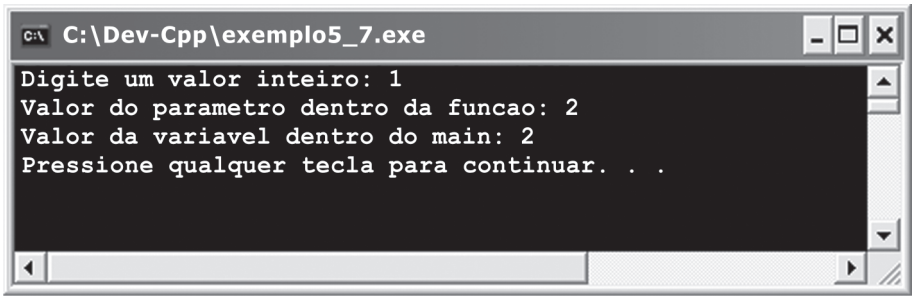


Figura 44 – Resultado da Execução do Exemplo 5

70. Refaça os exercícios 67, 68 e 69, utilizando protótipos de função.

7.5. FUNÇÕES RECURSIVAS

Funções recursivas são funções que contêm em seu corpo chamadas para si próprias. Em outras palavras, uma função *f* é dita recursiva se no corpo dessa função há chamadas à própria função *f*.

A implementação de alguns algoritmos fica muito mais fácil usando-se recursividade. Quando uma função é chamada recursivamente, cria-se um ambiente local para cada chamada. As variáveis locais de chamadas recursivas são independentes entre si, como se estivéssemos chamando funções diferentes.

Exemplo 6: A série de Fibonacci é muito conhecida na matemática. Por definição, o primeiro e o segundo termos da série de Fibonacci são o número 1. A partir do terceiro, todos os termos são a soma dos dois anteriores. Dessa forma, a série de Fibinacci é: 1, 1, 2, 3, 5, 8, 13, 21, 34...

Em outras palavras, podemos definir um termo *n* da série de Fibonacci, como:

- 1, se *n* for 1 ou 2. Ou seja, se for o primeiro ou o segundo termos.
- *Fibonacci (n-1) + Fibonacci (n-2)* se estivermos falando de um termo maior que o segundo.

Por exemplo, se quisermos o quarto termo da série, sabemos que este será a soma do terceiro termo com o segundo. O segundo sabemos que é 1. Mas, qual é o terceiro? Pela nossa definição, o terceiro será o segundo mais o primeiro, ou seja, o terceiro será dois. Assim, o quarto termo será 2 mais 1, ou seja, 3.

Façamos, então, como exemplo 6, um programa para encontrar o *enésimo* termo da sequência de Fibonacci, utilizando uma função recursiva:

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

//Função recursiva que calcula o enésimo termo da série de Fibonacci
int Fibonacci (int n) {
    //Se o termo for o primeiro ou o segundo, será 1
    if (n<=2)
       return 1;
    //Se o termo for maior que dois, será a soma de Fibonacci de n-1
    //com o Fibonacci de n-2
    else
       return (Fibonacci (n-1) + Fibonacci (n-2);
}

int main () {
    int n, termo;
    printf("Digite o numero do termo da serie de Fibonacci desejado: ");
    scanf("%d", &n);
    termo = Fibonacci (n);
    printf("O termo %d da serie de Fibonacci e %d \n", n, termo);
    system("pause");
}
```

Figura 45 – Exemplo 6

A figura 46 exibe o resultado da execução do programa quando solicitado o sétimo termo da série.

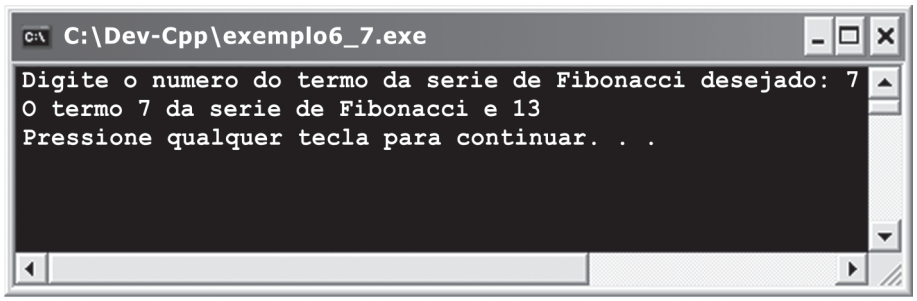


Figura 46 – Resultado da execução do exemplo 6

71. No capítulo 5 fizemos o exercício 54, no qual, dado um número qualquer, calculávamos o fatorial desse número. Para fazer esse exercício utilizamos uma estrutura de laço. Agora, refaça esse exercício utilizando uma função recursiva para calcular o fatorial do número dado. Na função main você deve solicitar a digitação do número do qual será calculado o fatorial, e chamar a função de cálculo do fatorial que deverá receber o número como parâmetro e retornar o fatorial do mesmo número para que a função principal o imprima. Vale lembrar que, dado um número n, o fatorial (n!) é:
    - 1, se n igual a 1;
    - n * fatorial (n-1), se n > 1.

    Assim, por exemplo, fatorial (4) = 4*fatorial (3) = 4*3*fatorial (2) = 4*3*2*fatorial (1) = 4*3*2*1 = 24


Anotar

REFERÊNCIAS

FARRER, Harry. **Algoritmos Estruturados.** Rio de Janeiro: LTC, 1999.

FORBELLONE, A. L. V., EBERSPÄCHER, H. F., **Lógica de Programação – A construção de algoritmos e estrutura de dados**. São Paulo: Makron Books, 2005

KERNIGHAN Brian W., **C Linguagem de Programação Padrão ANSI**. Rio de Janeiro: Elsevier, 1989.

LAUREANO, M. **Programando em C.** Rio de Janeiro: Brasport, 2005.

MIZRAHI, Victorine V. **Treinamento em Linguagem C – Curso Completo – Módulo 1**. São Paulo: Mc Graw Hill, 1990.

MORAES, P.S. **Curso básico de lógica de programação**, 2000; disponível em http://www.siban.com.br/destaque/21_carta.pdf

SANT'ANNA, S.R., **Programação I**, Vitória: CEFETES, 2007

SCHILDT, Herbert. **C Completo e Total**. São Paulo: Pearson, 2006.

TANENBAUM, Andrew S. **Sistemas Operacionais**. Porto Alegre: Bookman, 2000.