

LINGUAGEM E TÉCNICAS DE PROGRAMAÇÃO



PROFESSORES

Dra. Gislaine Camila Lapasini Leal
Me. Pietro Martins de Oliveira



ACESSO AQUI
O SEU LIVRO
NA VERSÃO
DIGITAL!

DIREÇÃO UNICESUMAR

Reitor Wilson de Matos Silva **Vice-Reitor** Wilson de Matos Silva Filho **Pró-Reitor de Administração** Wilson de Matos Silva Filho **Pró-Reitor Executivo de EAD** William Victor Kendrick de Matos Silva **Pró-Reitor de Ensino de EAD** Janes Fidélis Tomelin **Presidente da Mantenedora** Cláudio Ferdinandi

NEAD - NÚCLEO DE EDUCAÇÃO A DISTÂNCIA

Diretoria Executiva Chrystiano Mincoff, James Prestes, Tiago Stachon **Diretoria de Design Educacional** Débora Leite **Diretoria de Graduação** Kátia Coelho **Diretoria de Permanência** Leonardo Spaine **Diretoria de Pós-graduação, Extensão e Formação Acadêmica** Bruno Jorge **Head de Produção de Conteúdos** Celso Luiz Braga de Souza Filho **Gerência de Produção de Conteúdo** Diogo Ribeiro Garcia **Gerência de Projetos Especiais** Daniel Fuverki Hey **Supervisão do Núcleo de Produção de Materiais** Nádila Toledo **Supervisão de Projetos Especiais** Yasminn Zagonei

Coordenador(a) de Conteúdo

Danillo Xavier Saes

Projeto Gráfico e Capa

Arthur Cantareli, Jhonny Coelho
e Thayla Guimarães

Editoração

Sabrina Novaes

Design Educacional

Kaio Vinícius Cardoso Gomes

Revisão Textual

Silvia Caroline Gonçalves

Ilustração

André Azevedo

Fotos

Shutterstock

FICHA CATALOGRÁFICA

C397 CENTRO UNIVERSITÁRIO DE MARINGÁ.

Núcleo de Educação a Distância. **LEAL**, Gislaine Camila Lapasini. **OLIVEIRA**, Pietro Martins de.

Linguagem e Técnicas de Programação.

Gislaine Camila Lapasini Leal; Pietro Martins de Oliveira.

Maringá - PR.: UniCesumar, 2023.

208 p.

"Graduação - EaD".

1. Algoritmos 2. Lógica 3. Programação. EaD. I. Título.

CDD - 22 ed. 005.1

CIP - NBR 12899 - AACR/2

ISBN 978-85-459-2427-2

Bibliotecário: João Vivaldo de Souza CRB- 9-1679



NEAD - Núcleo de Educação a Distância

Av. Guedner, 1610, Bloco 4 Jd. Aclimação - Cep 87050-900 | Maringá - Paraná

www.unicesumar.edu.br | 0800 600 6360

AVALIE ESTE LIVRO!



ACESSE O QR CODE



CRIAR MOMENTOS DE APRENDIZAGENS
INESQUECÍVEIS É O NOSSO OBJETIVO E POR ISSO,
GOSTARÍAMOS DE SABER COMO FOI SUA EXPERIÊNCIA.

Conta para nós! leva menos de 2 minutos. Vamos lá?!

DIGITE O CÓDIGO

02511488

AA

RESPOnda A
PESQUISA

?

...

>>

Dra. Gislaine Camila Lapasini Leal

Doutora em Engenharia Elétrica e Informática Industrial pela Universidade Tecnológica Federal do Paraná. Mestre em Ciência da Computação pela Universidade Estadual de Maringá (2010). Graduada em Engenharia de Produção – Software pela Universidade Estadual de Maringá (2007) e em Processamento de Dados pelo Centro Universitário de Maringá (2004). Atualmente, é professora do Departamento de Engenharia de Produção da Universidade Estadual de Maringá, atuando na área de Apoio à Tomada de Decisão (Pesquisa Operacional e Gestão de Tecnologia da Informação). Tem interesse nas seguintes áreas: Otimização, Apoio à Tomada de Decisão em Sistemas de Produção, Startups de Software, Melhoria da Qualidade e Produtividade no Desenvolvimento de Software e Melhoria de Processos.

<http://lattes.cnpq.br/7810321373328408>

Me. Pietro Martins de Oliveira

Possui graduação em Engenharia de Computação pela Universidade Estadual de Ponta Grossa (2011). É mestre em Ciência da Computação na área de Visão Computacional pela Universidade Estadual de Maringá (2015). Atuou como analista de sistemas e programador nas empresas Siemens Enterprise Communications (Centro Internacional de Tecnologia de Software — CITS, 2011-2012) e Benner Saúde Maringá (2015). Experiência como docente e coordenador dos cursos de Bacharelado em Engenharia de Software, Sistemas de Informação e Engenharia de Produção.

<http://lattes.cnpq.br/1793084774574585>

LINGUAGEM E TÉCNICAS DE PROGRAMAÇÃO

Caro(a) estudante! Bem-vindo(a) à disciplina de Linguagem e Técnicas de Programação. Sou a professora Gislaine Camila e, juntamente com o professor Pietro Martins, aprenderemos, nesta disciplina, a utilizar a linguagem C na construção de nossos programas. Para tanto, retomaremos alguns conceitos vistos na disciplina de Algoritmos e Lógica de Programação I.

Nesta disciplina, você aprenderá os conceitos iniciais da linguagem de programação C, a qual vem se popularizando por ser uma linguagem de propósito geral e não vinculada a um hardware específico ou a qualquer outro sistema.

Apresentamos a você o livro que norteará seus estudos nesta disciplina, auxiliando no aprendizado da linguagem C. Em cada unidade, serão apresentados exemplos de programas em C. É importante que você compile cada um desses programas, gerando o executável e verificando o funcionamento de cada um deles. Apenas a leitura dos exemplos não é suficiente – o aprendizado requer prática.

Na Unidade 1, veremos o histórico da linguagem C, suas características e os conceitos iniciais sobre programação, destacando as etapas para a criação de um programa bem como sua estrutura. Estudaremos os tipos de dados disponíveis na linguagem C, como nomear identificadores, declarar variáveis e constantes, realizar operações de atribuição, entrada e saída de dados. Conheceremos, também, as palavras reservadas dessa linguagem, os operadores e as funções intrínsecas. A partir destes conteúdos, iniciaremos a construção de nossos primeiros programas em C.

Na Unidade 2, aprenderemos a construir programas com desvio de fluxo, isto é, imporremos condições para a execução de determinada instrução ou um conjunto de instruções. Trataremos de como construir programas em C, utilizando a estrutura condicional simples, a composta e a case.

A Unidade 3 abordará a construção de programas com repetição de determinado trecho de código sem a necessidade de reescrevê-lo várias vezes. Estudaremos as estruturas de repetição disponíveis na linguagem C: estrutura for, estrutura **while** e estrutura do **while**. Discutiremos as diferenças de cada uma delas e como utilizá-las.

APRESENTAÇÃO DA DISCIPLINA

Na Unidade 4, serão apresentados os conceitos de estruturas de dados homogêneas (vetores e matrizes) e estruturas de dados heterogêneas (*structs*). Estas estruturas permitem agrupar diversas informações em uma única variável. Aprenderemos como declará-las e manipulá-las nas operações de entrada, atribuição e saída. Trataremos de questões relacionadas à pesquisa de um elemento em um vetor e à ordenação de um vetor segundo algum critério. Estudaremos, também, como manipular as cadeias de caracteres (*strings*) e conheceremos as funções disponíveis na linguagem C que nos permitem concatenar, comparar, verificar tamanho, converter os caracteres para maiúsculo/minúsculo e outros.

Por fim, na Unidade 5, trabalharemos com a modularização de nossos programas, utilizando funções. Abordaremos os conceitos relacionados ao escopo de variáveis e à passagem de parâmetros por valor e por referência, protótipo de função e recursividade. Estudaremos as funções disponíveis para a manipulação de arquivos que nos permitem abrir um deles, verificar erro durante a abertura, verificar fim de arquivo, fechá-lo, ler e gravar caractere, cadeia de caracteres e demais tipos de dados. Serão apresentados programas que ilustram o funcionamento de cada uma dessas funções.

Em cada unidade deste livro, você encontrará indicações de leitura complementar, as quais enriquecerão o seu conhecimento e apresentarão mais exemplos de programas em C. Além disso, serão apresentadas atividades de autoestudo, as quais permitem que você coloque em prática os conhecimentos, e exercícios de fixação, que apresentam exercícios resolvidos para auxiliar no aprendizado e sanar eventuais dúvidas.

Desejamos a você um bom estudo!

ÍCONES



pensando juntos

Ao longo do livro, você será convidado(a) a refletir, questionar e transformar. Aproveite este momento!



explorando Ideias

Neste elemento, você fará uma pausa para conhecer um pouco mais sobre o assunto em estudo e aprenderá novos conceitos.



quadro-resumo

No fim da unidade, o tema em estudo aparecerá de forma resumida para ajudar você a fixar e a memorizar melhor os conceitos aprendidos.



conceituando

Sabe aquela palavra ou aquele termo que você não conhece? Este elemento ajudará você a conceituá-la(o) melhor da maneira mais simples.



conecte-se

Enquanto estuda, você encontrará conteúdos relevantes online e aprenderá de maneira interativa usando a tecnologia a seu favor.



Quando identificar o ícone de QR-CODE, utilize o aplicativo Unicesumar Experience para ter acesso aos conteúdos online. O download do aplicativo está disponível nas plataformas:



Google Play



App Store

CONTEÚDO

PROGRAMÁTICO

UNIDADE 01

10

CONCEITOS INICIAIS

UNIDADE 02

49

ESTRUTURA
CONDICIONAL

UNIDADE 03

79

ESTRUTURAS DE
REPETIÇÃO

UNIDADE 04

105

VETORES, *STRINGS*,
MATRIZES E *STRUCTS*

UNIDADE 05

143

FUNÇÕES E
ARQUIVOS

FECHAMENTO

174

CONCLUSÃO GERAL



CONCEITOS INICIAIS

PROFESSORES

Dra. Gislaine Camila Lapasini Leal
Me. Pietro Martins de Oliveira

PLANO DE ESTUDO

A seguir, apresentam-se as aulas que você estudará nesta unidade:

- A linguagem C e os conceitos iniciais de programação
- A estrutura de um programa em C
- Os identificadores, os tipos de dados e as palavras reservadas
- As variáveis, as constantes, as expressões e os operadores
- As funções intrínsecas e a atribuição
- A entrada e a saída de dados em um programa
- Exercícios resolvidos

OBJETIVOS DE APRENDIZAGEM

- Compreender os conceitos iniciais de programação e a estrutura de um programa em C
- Estudar a estrutura de um programa em C
- Compreender o que são e como utilizar os identificadores, os tipos de dados e as palavras reservadas
- Estudar as variáveis, as constantes e as expressões e os operadores
- Compreender as funções intrínsecas e a atribuição
- Estudar a entrada e a saída de dados por meio da construção de programas
- Estudar os exercícios resolvidos

INTRODUÇÃO



Nesta primeira unidade, você será introduzido(a) ao universo da linguagem de programação C, uma linguagem poderosa, popular e que é utilizada em uma vasta gama de aplicações, como em sistemas embarcados, sistemas operacionais, dentre outras.

Primeiramente, você conhecerá o histórico da linguagem C, suas características, seus pontos fortes e fracos e compiladores disponíveis para ela. Estudará os conceitos básicos relacionados à programação que possibilitam entender como um código-fonte é convertido em um programa executável. Além disso, terá contato com a interface de desenvolvimento e com o compilador que adotaremos na construção de nossos programas.

Estudaremos a estrutura básica de um programa em C e os elementos que compõem esta linguagem. Ao construir nossos programas, precisamos guardar algumas informações a respeito do problema. Para isso, veremos o conceito de variáveis e constantes e sua sintaxe em C. Conheceremos tipos de dados disponíveis na linguagem, expressões e operadores, funções intrínsecas e como utilizá-los em nossos programas. Para obter dados dos usuários e mostrar mensagens e resultados de processamento, estudaremos as funções relacionadas à entrada de dados, que nos permitem interagir como usuário; o comando de atribuição, que possibilita atribuir valor às variáveis; e as funções de saída de dados, que permitem o envio de mensagens e a exibição dos resultados do processamento na saída padrão do dispositivo computacional (geralmente, o monitor, a tela).

Ao final desta unidade, você terá adquirido conhecimento da estrutura de um programa, de variáveis, constantes, palavras reservadas da linguagem C, expressões e operadores, funções intrínsecas, comando de atribuição, função de entrada e de saída de dados. Com estes conceitos, você saberá construir os primeiros programas em C. Vamos lá?



1 A LINGUAGEM C E OS CONCEITOS INICIAIS de Programação

A linguagem C foi concebida e implementada, inicialmente, para o sistema operacional Unix, na década de 70, por Dennis Ritchie, nos Laboratórios Bell da companhia AT&T (KERNIGHAN; RITCHIE, 1988). C é uma linguagem de programação de propósito geral, com uma sintaxe muito compacta e que permite a combinação de operadores de diferentes tipos. Além disso, não está vinculada a um hardware específico ou a qualquer outro sistema, de modo que é fácil escrever programas que serão executados sem mudanças em qualquer máquina que suporta C (KERNIGHAN; RITCHIE, 1988).

Segundo Rocha (2006), a principal característica da C é que ela combina as vantagens de uma linguagem de alto nível com a eficiência da linguagem de montagem (linguagem de máquina ou *assembly*). Em C, é possível realizar operações aritméticas sobre ponteiros e operações sobre palavras binárias.

Podemos dizer que esta liberdade oferecida pela linguagem C é uma faca de dois gumes, pois, ao passo que permite programadores experientes de escreverem códigos mais compactos e eficientes, possibilita que programadores inexperientes realizem construções sem sentido, as quais são aceitas como válidas pelo compilador. Deste modo, ao construir programas utilizando C, devemos ficar atentos às construções da linguagem (ROCHA, 2006).

Ascencio e Campos (2010) destacam que, durante alguns anos, o padrão da linguagem C foi fornecido com a versão do sistema operacional Unix. No entanto

com a popularização dos microcomputadores, foram criadas várias implementações da linguagem C, o que ocasionou várias discrepâncias. Em 1983, o ANSI (*American National Standards Institute*) criou um comitê para definir um padrão que guiasse todas as implementações da linguagem C.



conecte-se

Para entender como instalar o Bloodshed Dev-C++ no Windows, acesse o vídeo disponível em:

<https://youtu.be/amwKFSFLsB4>

Fonte: os autores.



Na literatura, podemos encontrar diversos compiladores C, sendo, o principal deles, o *GNU Compiler Collection* (GCC). Para além do simples compilador, existem diversos Ambientes Integrados de Desenvolvimento (*Integrated Development Environment* — IDE), que são softwares mais robustos para criação de projetos, edição de código-fonte, compilação, teste, depuração (*debug*) etc. Algumas das IDEs mais conhecidas são: Dev-C++, Microsoft Visual Studio Code, Code Blocks etc. Em nossa disciplina de Algoritmos e Lógica de Programação II, adotaremos o Dev-C++ (conforme a figura a seguir):

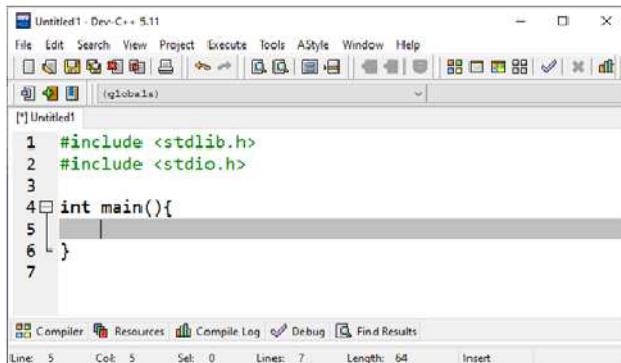


Figura 1 - Interface Dev-C++ / Fonte: os autores.

Conceitos iniciais de programação

A programação inicia-se com a escrita do programa (código-fonte) e encerra-se com a geração de um programa executável. A escrita deve ser realizada em um editor de textos. Após a criação do código contendo a lógica do programa, temos

que compilá-lo. O processo de compilação analisa o código, do ponto de vista sintático, e o converte para um código - objeto, que é a versão em linguagem de máquina do programa. Se o programa possui chamada às funções de bibliotecas, o *linker* (ligador) reúne o programa - objeto com as bibliotecas referenciadas e gera o código executável (arquivo binário) (ROCHA, 2006).

A Figura 2 ilustra o processo de criação de um programa, desde a criação do código-fonte até a geração de um programa executável.

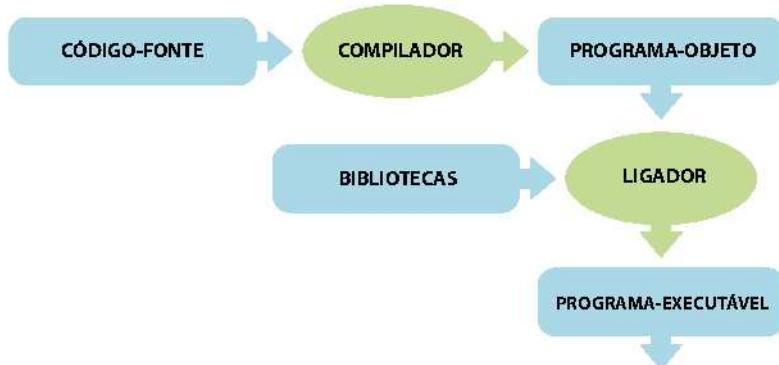


Figura 2 - Etapas para a criação de um programa / Fonte: adaptada de Rocha (2006).

Em nossa disciplina, escreveremos nossos programas utilizando o Dev-C++. Na Figura 3, são mostrados os menus em que as operações compilar, ligar e executar são realizadas.

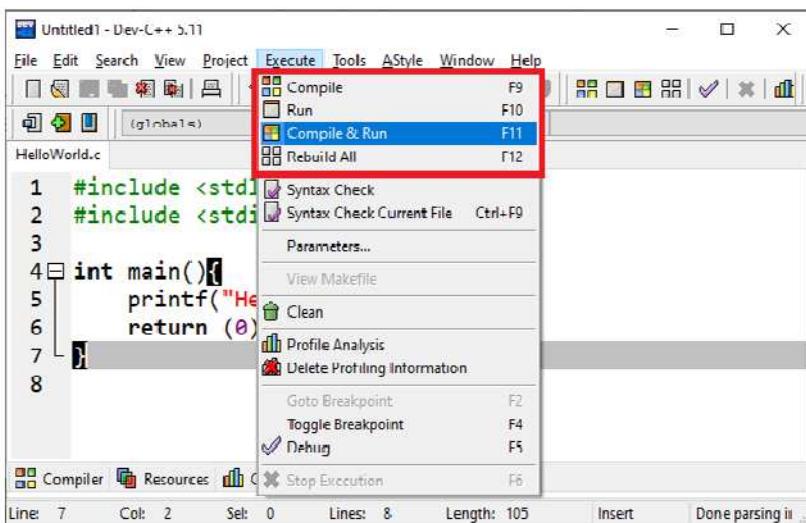


Figura 3 - Dev-C++ / Fonte: os autores.



A ESTRUTURA DE UM PROGRAMA EM C

2

Kernighan e Ritchie (1988) destacam que a única maneira de aprender uma nova linguagem de programação é escrevendo programas nela. Com isso, para entender a estrutura de um programa em C, construiremos nosso primeiro programa, o famoso “Hello, world” (quadro a seguir):

```
01  #include <stdio.h>
02  int main()
03  {
04      printf("Hello, World");
05      return (0);
06  }
```

Quadro 1 - Programa *Hello, World* / Fonte: os autores.

Fique tranquilo(a)! Analisaremos o programa anterior linha a linha e entenderemos o que cada um destes elementos significa.

Na primeira linha, temos a instrução `#include <stdio.h>`, a qual indica ao compilador que, durante o processo de compilação e linkagem, deve-se incluir o

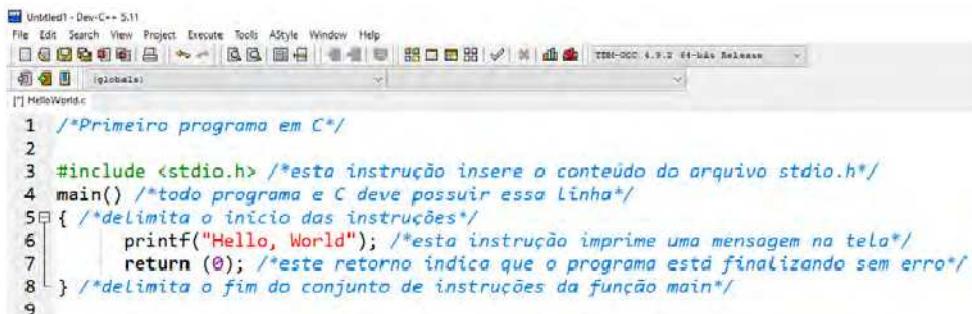
conteúdo do arquivo predefinido (é um arquivo que já existe — uma biblioteca com funcionalidades pré-programadas), chamado `stdio.h`. Este arquivo chama-se arquivo de cabeçalho e contém declarações de funções para entrada e saída de dados. Portanto, não podemos esquecer de inserir esta linha em nossos programas.

Em seguida, temos a instrução `main()`, que identifica a função principal de qualquer programa escrito em linguagem C, denominada `main`. Os programas em C são formados por chamadas de função. Obrigatoriamente, todo programa deve possuir uma função `main`, a qual é a primeira a ser chamada quando o programa é executado. Note que o conteúdo da função é delimitado por chaves, de modo análogo ao início e ao fim dos algoritmos escritos em pseudocódigo. Isto quer dizer que o conteúdo entre as chaves será executado, sequencialmente, quando o programa for executado.

Dentro das chaves, temos duas instruções: a primeira delas é o `printf`, uma função previamente definida no arquivo de cabeçalho `stdio.h`. Neste caso, o `printf` é responsável por imprimir, na tela, a sequência de caracteres “Hello, World”. A última linha, `return(0)`, indica o valor de retorno da função. No caso, 0, por convenção, indica que o programa terminou sem erros. Observe que, ao final de cada instrução, há um ponto e vírgula, isto é, a grande maioria dos comandos em C terminam com “;”.

Agora que vimos cada um dos elementos, você deve estar se perguntando como executar isso. Para visualizar nosso programa, precisamos escrever o código no Dev-C++, conforme Figura 4. Observe que, além do código apresentado no Quadro 1, há outras informações que estão entre `/* */`. Este comando (barra + asterisco + asterisco + barra) é utilizado para escrever comentários em nosso código, isto é, o compilador desconsidera qualquer coisa que esteja entre estes dois pares de símbolos. O compilador não considera comentários como comandos executáveis, apenas texto de “anotação”. Um comentário pode ter mais de uma linha.

Os comentários são textos que podem ser inseridos com o objetivo de documentá-lo. Em nossos programas, adotaremos, como padrão, a escrita de comentários. Eles nos auxiliam a entender o funcionamento dos programas, além de serem uma boa prática de programação, desde que utilizados com sabedoria.



```

1 /*Primeiro programa em C*/
2
3 #include <stdio.h> /*esta instrução insere o conteúdo do arquivo stdio.h*/
4 main() /*todo programa em C deve possuir essa Linha*/
5 { /*delimita o inicio das instruções*/
6     printf("Hello, World"); /*esta instrução imprime uma mensagem na tela*/
7     return (0); /*este retorno indica que o programa está finalizando sem erro*/
8 } /*delimita o fim da conjunto de instruções da função main*/
9

```

Figura 4 - Primeiro programa em C – Código / Fonte: os autores.

Após escrever o código do programa, temos que compilá-lo (Atalho pelo teclado: F9). É neste momento que serão realizadas as verificações de sintaxe. E, para visualizar o resultado do programa, basta executá-lo. No caso do Dev-C++, podemos ir ao menu Execute > Run ou pressionar a tecla F10.

Ao executar o código, temos como resultado a impressão, na tela, da cadeia de caracteres “Hello, World”, conforme pode ser visualizado na figura a seguir:



Figura 5 - Primeiro programa em C – Saída (Execução) / Fonte: os autores.



conecte-se

Para conhecer a linguagem C mais a fundo, acesse o texto disponível em:
http://www.eletrica.ufpr.br/graduacao/noturno/docs/te207/apostila_C.pdf
Fonte: os autores.



A estrutura geral de um programa em C é apresentada no quadro a seguir:

01	<inclusão de bibliotecas>
02	int main()
03	{
04	<conjunto de instruções>
05	return (0);
06	}

Quadro 2 - Estrutura geral de um programa em C / Fonte: os autores.

Quando quisermos criar nossos programas em linguagem C, basta substituirmos os campos inclusão de bibliotecas e conjunto de instruções pelos respectivos comandos pertinentes.

AULA

OS IDENTIFICADORES, OS TIPOS DE DADOS e as Palavras Reservadas

Os identificadores consistem nos nomes que utilizamos para representar variáveis, constantes, tipos, funções e rótulos do programa. Um identificador é uma sequência de uma ou mais letras, dígitos ou sublinhas (caractere *underline* ou *underscore* “_”), que começa com uma letra ou um sublinhado. Em geral, evita-se iniciar um identificador com sublinhas, pois este tipo de notação, geralmente, é reservado para o compilador (PAPPAS; MURRAY, 1991).

A linguagem C é *case sensitive*, isto é, o compilador considera letras maiúsculas e minúsculas como caracteres distintos, diferenciando caixa alta de caixa baixa. Os comandos (palavras reservadas) em C só podem ser escritos em minúsculas, senão o compilador os interpretará como variáveis. O Quadro 3 apresenta alguns exemplos de identificadores válidos e inválidos.

Identificadores válidos	Identificadores inválidos
A	2 ^a
a	b@
media	media idade
altura2	x*y
media_idade	#media
x36	idade!

Quadro 3 - Exemplo de identificadores / Fonte: os autores.

Devemos lembrar que, em C, o identificador “A” não é o mesmo que o identificador “a”; cada um é um identificador único. Caso A e a fossem utilizados para nomear variáveis, seriam considerados variáveis distintas e independentes uma da outra.

Os Tipos de dados

Na linguagem C, as informações podem ser representadas por sete tipos básicos de dados: `char`, `int`, `float`, `double`, `enum`, `void` e `pointer` (ponteiro) (PAPPAS; MURRAY, 1991).

O tipo `char` é utilizado para representar caracteres simples e até *strings* (cadeia de caracteres). O tipo `int` são dados numéricos que não possuem componentes decimais ou fracionários. O tipo `float`, valores em ponto flutuante, são números que têm componente decimal ou fracionário. O tipo `double` são valores em ponto flutuante de precisão dupla, que apresentam alcance mais extenso. O tipo `enum`, dados enumerados, possibilitam os tipos definidos pelo usuário. O tipo `void` significa valores que ocupam 0 bits e não possuem valor algum, indicando uma espécie de “ausência de tipo”. O `pointer`, apesar de não ser uma palavra reservada, representa um dado especial, que não contém uma informação propriamente dita, mas sim, uma localização de memória (endereço de memória) que, por sua vez, contém o dado verdadeiro.

A partir dos tipos básicos, podem ser definidos outros tipos de dados utilizando modificadores. No Quadro 4, são apresentadas informações relativas à faixa de valores e ao tamanho aproximado dos tipos de dados. Observe que foram aplicados os modificadores `unsigned`, `short` e `long` aos tipos básicos.

O modificador `unsigned` é utilizado para declarar dados numéricos como sem sinal (apenas valores não negativos), duplicando, assim, a gama de valores que pode ser representada. O modificador `short` reduz a capacidade de armazenamento, enquanto o modificador `long` aumenta a capacidade.

Tipo	Faixa de valores	Tamanho (aproximado)
<code>char</code>	-128 a 127	8 bits
<code>unsigned char</code>	0 a 255	8 bits
<code>int</code>	-32.768 a 32.767	16 bits
<code>unsigned int</code>	0 a 65.535	16 bits
<code>short int</code>	-32.768 a 32.767	16 bits
<code>long</code>	-2.147.483.648 a 2.147.483.647	32 bits
<code>unsigned long</code>	0 a 4.294.967.295	32 bits
<code>float</code>	3.4×10^{-38} a 3.4×10^{38}	32 bits
<code>double</code>	1.7×10^{-308} a 1.7×10^{308}	64 bits
<code>long double</code>	3.4×10^{-49328} a 1.1×10^{4932}	80 bits

Quadro 4 - Tipos de dados e faixa de valores / Fonte: Ascencio e Campos (2010).

Ascencio e Campos (2010) apontam que a faixa de valores e o tamanho podem variar de acordo com o compilador. Os valores descritos anteriormente estão em conformidade com o padrão C ANSI.

As Palavras reservadas

As palavras reservadas, também conhecidas como palavras-chave, são identificadores que possuem uso específico para a linguagem C, ou seja, têm significados especiais.

Palavras reservadas			
asm	template	do	register
catch	this	double	return
clas	virtual	else	short
delete	_cs	enum	signed
_export	_ds	extern	sizeof
frient	_es	far	static
inline	_ss	float	struct
_loadds	auto	for	switch
new	break	goto	typedef
operator	case	huge	union
private	catch	if	unsigned
protected	cdecl	int	void
public	char	interrupt	volatile
_regparam	const	long	while
_saveregs	continue	near	
_seg	default	pascal	

Quadro 5 - Algumas palavras reservadas da linguagem C / Fonte: os autores.

No quadro anterior, são destacadas as palavras reservadas da linguagem C. Lembrar-se: não podemos utilizar, como identificador, uma palavra reservada.



AS VARIÁVEIS, AS CONSTANTES, as Expressões e os Operadores

Em nossos programas, precisamos armazenar algumas informações e, para isso, utilizamos as variáveis. Uma variável é um espaço na memória principal do computador que pode conter diferentes valores a cada instante de tempo (LOPES; GARCIA, 2002).

Na linguagem C, as variáveis são declaradas após a especificação de seus tipos, sendo os tipos mais utilizados: `int`, `float` e `char`. Note que, em C, não existe o tipo de dados `boolean`, pois considera verdadeiro qualquer valor diferente de 0. Além disso, não há um tipo especial para armazenar cadeia de caracteres (**strings**), sendo utilizado um vetor que contém vários elementos do tipo `char` para tal fim (ASCENCIO; CAMPOS, 2010).

A sintaxe para declaração de variáveis é dada por:

```
<tipo> <identificador>;
```

No quadro a seguir, são apresentados exemplos de declaração de variáveis. Atente para o fato de que a declaração de variáveis é seguida de ; (ponto e vírgula). Observe, também, que podemos declarar, em uma mesma linha, diversas variáveis do mesmo tipo.

Declaração	
<code>int quantidade;</code>	Declara uma variável chamada quantidade, que pode armazenar um valor inteiro.
<code>float total;</code>	Declara uma variável chamada total, que pode armazenar um valor real.
<code>float valor, total;</code>	Declara duas variáveis denominadas valor e total, que podem armazenar valor real.
<code>char sexo;</code>	Declara uma variável denominada sexo, que pode armazenar um caractere.
<code>char endereco[30];</code>	Declara uma variável denominada endereço, que pode armazenar até 30 caracteres.

Quadro 6 - Exemplos de declaração de variáveis / Fonte: os autores.

Constantes

Uma constante armazena informações que não variam com o tempo, ou seja, o seu conteúdo é um valor fixo. Em C, podemos definir constantes por meio da seguinte sintaxe:

```
#define <identificador> <valor>
```

Observe que, na definição de constantes, **não** utilizamos o ; no final.

Expressões e operadores

As expressões estão diretamente relacionadas ao conceito de fórmula matemática, em que um conjunto de variáveis e constantes relaciona-se por meio de operadores (LOPES; GARCIA, 2002).

As expressões **aritméticas** são aquelas em que o resultado consiste em um valor numérico. Desta forma, apenas operadores aritméticos e variáveis numéricas (`int`, `float` e/ou `double`) deveriam ser utilizadas em expressão deste tipo. O Quadro 7 apresenta os operadores aritméticos da linguagem C, destacando suas representações e forma de uso.

Operação	Operador	Significado
Soma	+	Utilizado para efetuar a soma de duas ou mais variáveis. Dada uma variável A e outra B, temos que a soma delas é representada por $A + B$.
Subtração	-	Simboliza a subtração do valor de duas variáveis. Supondo uma variável A e B, a diferença entre elas é dada por: $A - B$.
Multiplicação	*	O produto entre duas variáveis A e B é representado por $A * B$.
Divisão	/	A divisão entre duas variáveis A e B é dada por: A/B . Em relação à divisão, é importante lembrar que não existe divisão por zero.
Resto de divisão	%	Usado quando se deseja encontrar o resto da divisão entre duas variáveis A e B. A representação é dada por $A \% B$. Supondo $A = 3$ e $B = 2$, temos que $A \% B = 1$, uma vez que 3 divididos por 2 resulta em 1, com resto igual a 1.

Quadro 7- Operadores aritméticos / Fonte: os autores.

Em C, temos, também, os operadores aritméticos de atribuição (Quadro 8), que são utilizados para representar, de maneira sintética, uma operação aritmética, seguida de uma operação de atribuição (ASCENCIO; CAMPOS, 2010).

Operador	Exemplo	Explicação
<code>+=</code>	<code>x += y</code>	Equivale a $x = x + y$.
<code>-=</code>	<code>x -= y</code>	Equivale a $x = x - y$.
<code>*=</code>	<code>x *= y</code>	Equivale a $x = x * y$.
<code>/=</code>	<code>x /= y</code>	Equivale a $x = x / y$.
<code>%=</code>	<code>x %= y</code>	Equivale a $x = x \% y$.
	<code>x++</code>	Equivale a $x = x + 1$.
<code>++</code>	<code>y = ++x</code>	Equivale a $x = x + 1$ e, depois, $y = x$.
	<code>y = x++</code>	Equivale a $y = x$ e, depois, $x = x + 1$.
	<code>x--</code>	Equivale a $x = x - 1$.
<code>--</code>	<code>y = --x</code>	Equivale a $x = x - 1$ e, depois, $y = x$.
	<code>y = x--</code>	Equivale a $y = x$ e, depois, $x = x - 1$.

Quadro 8 - Operadores matemáticos de atribuição / Fonte: os autores.

As expressões **relacionais** referem-se à comparação entre dois valores de um tipo básico. Tal comparação relacional, em linguagem C, pode resultar em verdadeiro ou falso. Os operadores relacionais são destacados no Quadro 9, em que é possível visualizar o operador, o símbolo associado e a forma de uso.

Operador	Símbolo	Exemplo
Igual	<code>==</code>	<code>A == 1</code>
Diferente	<code>!=</code>	<code>A != B</code>
Maior	<code>></code>	<code>A > 5</code>
Menor que	<code><</code>	<code>B < 12</code>
Maior ou igual a	<code>>=</code>	<code>A >= 6</code>
Menor ou igual a	<code><=</code>	<code>B <= 7</code>

Quadro 9 - Operadores relacionais / Fonte: os autores.

As expressões lógicas são aquelas cujo resultado consiste em um valor lógico verdadeiro ou falso. Neste tipo de expressão, podem ser usados os operadores relacionais, os operadores lógicos ou as expressões matemáticas.

No quadro a seguir, são descritos cada um dos operadores lógicos: conjunção, disjunção e negação.

Operador	Símbolo	Exemplo
Disjunção	<code> </code>	A disjunção entre duas variáveis resulta em um valor verdadeiro quando, pelo menos, uma das variáveis é verdadeira.
Conjunção	<code>&&</code>	A conjunção entre duas variáveis resulta em um valor verdadeiro somente quando as duas variáveis são verdadeiras.
Negação	<code>!</code>	A negação inverte o valor de uma variável. Se a variável A é verdadeira, então, a negação de A torna o valor da variável falso.

Quadro 10 - Operadores lógicos / Fonte: os autores.

Em uma expressão, podemos ter mais de um operador. Em situações em que há um único operador, a avaliação da expressão é realizada de forma direta. Quando há mais de um, é necessária a avaliação da expressão passo a passo, ou seja, um operador por vez.

Operador	Descrição	Associa pela	Precedência
<code>()</code>	Expressão de função	Esquerda	Mais alta
<code>++ --</code>	Incremento/decremento	Esquerda	
<code>!</code>	Negação	Direita	
<code>&</code>	Endereço	Direita	
<code>*</code>	Multiplicação	Esquerda	
<code>/</code>	Divisão	Esquerda	
<code>%</code>	Resto	Esquerda	
<code>+</code>	Adição	Esquerda	
<code>-</code>	Subtração	Esquerda	
<code><</code>	Menor que	Esquerda	
<code><=</code>	Menor ou igual	Esquerda	
<code>></code>	Maior que	Esquerda	
<code>>=</code>	Maior ou igual	Esquerda	
<code>==</code>	Igual	Esquerda	
<code>!=</code>	Diferente	Esquerda	
<code>&&</code>	Conjunção (E lógico)	Esquerda	
<code> </code>	Disjunção (Ou lógico)	Esquerda	
<code>? :</code>	Condisional	Direita	
<code>=, %=, +=, -=, *=, /=</code>	Atribuição	Esquerda	
<code>,</code>	Vírgula	Esquerda	Mais baixa

Quadro 11 - Níveis de precedência de operador / Fonte: adaptado de Pappas e Murray (1991).

No Quadro 11, é apresentado um resumo com os principais operadores da linguagem C, destacando desde a precedência mais alta até a mais baixa, e descreve como cada operador está associado (esquerda para direita ou direita para esquerda). Cabe ressaltar que os operadores entre linhas têm a mesma precedência.





AS FUNÇÕES INTRÍNSECAS e a atribuição

As funções intrínsecas são fórmulas matemáticas prontas que podemos utilizar em nossos programas. O quadro a seguir apresenta as principais funções da linguagem C, destacando o comando associado, um exemplo e o que esta linguagem faz.

Função	Exemplo	Objetivo
ceil	ceil(x)	Arredonda um número real para cima. Por exemplo, ceil(2.3) é 3.
cos	cos(x)	Calcula o cosseno de x. O valor de x deve estar em radianos.
exp	exp(x)	Obtém a constante de Euler elevada à potência x.
abs	abs(x)	Retorna o valor absoluto de x.
floor	floor(x)	Arredonda um número real para baixo. Por exemplo, floor(2.3) é 2.
log	log(x)	Retorna o logaritmo natural de x.
log10	log10(x)	Retorna o logaritmo de x na base 10.
modf	z=modf(x, &y)	Decompõe o número real, armazenado em x, em duas partes: y recebe a parte fracionária e z, a parte inteira.

Função	Exemplo	Objetivo
pow	pow (x, y)	Calcula a potência de x elevado a y.
sin	sin (x)	Calcula o seno de x.
sqrt	sqrt (x)	Calcula a raiz quadrada de x.
tan	tan (x)	Calcula a tangente de x.
M_PI	M_PI	Retorna o valor da constante trigonométrica π .

Quadro 12 - Funções intrínsecas da linguagem C (`math.h`) / Fonte: os autores.

Atribuição

O comando de atribuição é usado para conceder valores ou operações a variáveis. O símbolo utilizado para a atribuição é o = (sinal de igualdade). A sintaxe para atribuição é dada por:

`<identificador> = <expressão>`

O identificador representa a variável à qual será atribuído um valor, e o termo `<expressão>` é o conteúdo que será atribuído, podendo ser uma expressão aritmética ou lógico-relacional, uma outra variável, constante, etc. Alguns exemplos do uso do comando de atribuição podem ser visualizados no quadro a seguir.

Exemplo	
<code>x = 3;</code>	O valor 3 é atribuído à variável x.
<code>x = x + 5;</code>	É atribuído à variável x o valor da própria variável x acrescido em 5.
<code>x = 2.5;</code>	O valor 2.5 é atribuído à variável x.
<code>sexo = 'M';</code>	O valor 'M' é atribuído à variável denominada sexo.
<code>nome = "Maria";</code>	O valor "Maria" é atribuído à cadeia de caracteres chamada nome.

Quadro 13 - Exemplos de atribuição / Fonte: os autores.

Note que, na linguagem C, os caracteres simples são representados entre apóstrofos ('), e as cadeias de caracteres, entre aspas ("").



A ENTRADA E A SAÍDA DE DADOS em um programa

A entrada de dados permite receber os dados digitados pelo usuário por meio da entrada padrão do dispositivo computacional, geralmente, o teclado. Os dados recebidos são armazenados em variáveis. Na linguagem C, existem diversas funções para entrada de dados, algumas delas são `scanf` e `gets` (quadro a seguir) (ASCENCIO; CAMPOS, 2010).

Os comandos `gets` e `scanf` armazenam toda a cadeia de caracteres até que seja pressionada a tecla Enter.

Exemplo	
<code>gets(<variável>);</code>	Um ou mais caracteres digitados pelo usuário são armazenados na variável.
<code>scanf("<texto>", &<variável>);</code>	Um valor digitado pelo usuário é armazenado na variável .

Quadro 14 - Exemplos de entrada de dados / Fonte: adaptado de Ascencio e Campos (2010).

A função `scanf` é a mais utilizada, sendo sua sintaxe dada por:

`scanf("<expressão de controle>", <lista de variáveis>);`

O argumento <expressão de controle> deve ser escrito entre aspas e contém os especificadores de formato (Quadro 15), que indicam como os dados digitados devem ser armazenados. No argumento <lista de variáveis>, as variáveis devem ser separadas por vírgulas e cada uma delas deve ser precedida pelo operador de endereço (&). As variáveis usadas para receber valores por meio da função `scanf` deverão ser passadas pelos seus endereços. O operador de endereço indica o endereço da posição de memória para a variável (ROCHA, 2006).

Na leitura de cadeias de caracteres (*strings*), não se utiliza o operador de endereço (&), pois o identificador do vetor já é o endereço do primeiro elemento do vetor.

Código	Significado
%c	Leitura de um único caractere .
%d	Leitura de um número decimal inteiro.
%i	Leitura de um decimal inteiro.
%u	Leitura de um decimal sem sinal.
%e	Leitura de um número em ponto flutuante com sinal opcional.
%f	Leitura de um número em ponto flutuante com ponto opcional.
%g	Leitura de um número em ponto flutuante com expoente opcional.
%o	Leitura de um número em base octal.
%s	Leitura de uma <i>string</i> .
%x	Leitura de um número em base hexadecimal.
%p	Leitura de um ponteiro.

Quadro 15 - Especificadores de formato / Fonte: adaptado de Rocha (2006).

Tomemos como exemplo: para ler uma variável que armazena a idade, o uso da função `scanf` deve ser realizado do seguinte modo:

```
scanf ("%d", &idade);
```

Lembrando que, para utilizar os comandos de entrada de dados, é necessário incluir a biblioteca `stdio.h`, utilizando o comando `#include <stdio.h>;`

Saída de dados

A saída de dados permite mostrar estes aos usuários. Na linguagem C, geralmente, utilizamos a função `printf` para exibir resultados do processamento e mensagens. A sintaxe desta função é:

```
printf("<expressão de controle>", <lista de argumentos>);
```

O argumento `<expressão de controle>` pode conter mensagens que serão exibidas na tela e os especificadores de formato que indicam o formato que os argumentos devem ser impressos, além de caracteres especiais indicadores de formatação de texto, como é o caso do `\n`, `\t`, etc. A `<lista de argumentos>` pode conter identificadores de variáveis, expressões aritméticas ou lógico-relacionais e valores constantes.

Além dos especificadores de formato, podemos utilizar códigos especiais na saída de dados. Estes códigos são apresentados no quadro a seguir:

Código	Significado
<code>\n</code>	Nova linha
<code>\t</code>	Tab
<code>\b</code>	Retrocesso
<code>\"</code>	Aspas
<code>\\"</code>	Contrabarra
<code>\f</code>	Salta página de formulário
<code>\0</code>	Nulo

Quadro 16 - Códigos especiais / Fonte: adaptado de Pappas e Murray (1991).

O programa da Figura 6 apresenta alguns exemplos de uso da função `printf`. Para facilitar o entendimento, verifique o que é exibido, em vídeo, para cada uma das instruções.

The screenshot shows the Dev-C++ IDE interface. The title bar displays the path "C:\Users\spieri\Documents\Códigos Livro ALP 2\TestePrints.cpp - [Executing] - Dev-C++ 5.11". The menu bar includes File, Edit, Search, View, Project, Execute, Tools, AStyle, Window, Help. The toolbar contains various icons for file operations like Open, Save, Print, and Build. The status bar at the bottom right shows "TIFF-GCC 4.9.2 64-bit Release". The code editor window shows the following C code:

```

1 #include <stdio.h>
2 int main(){
3     printf("Estou aprendendo a programar em C");
4     printf("Estou lendo a %d unidade do livro", 1);
5     printf("%s é uma disciplina importante do curso", "Esta");
6     printf("%f", 57.35);
7     return (0);
8 }
9

```

Figura 6 - Entendendo a função printf – Código / Fonte: os autores.

A saída obtida com a execução do programa da Figura 6, é apresentada na Figura 7. Como podemos observar, as mensagens foram impressas em tela sem quebra de linha. Para imprimir cada mensagem em uma linha, devemos utilizar o comando especial (\n). Este código pode ser visualizado na Figura 8.

The terminal window shows the execution output of the program. The output is:

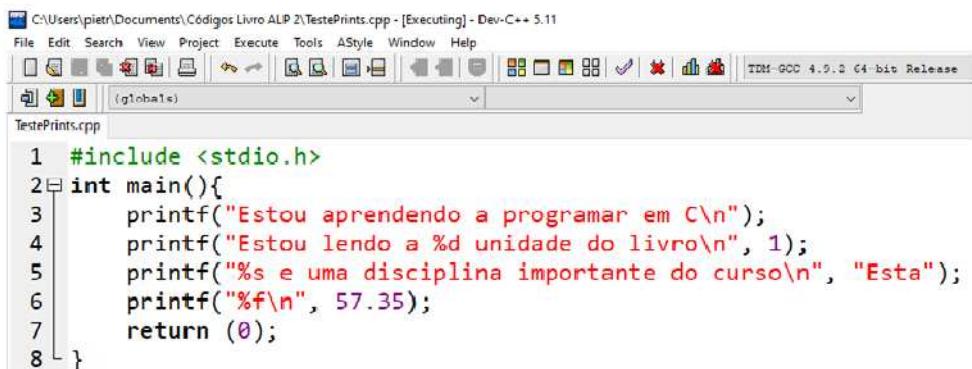
```

Estou aprendendo a programar em C
Estou
lendo a 1 unidade do livro
Esta é uma di
sciplina importan
te do curso
57.350000
-----
Process exited after 4.16 seconds with
return value 0
Pressione qualquer tecla para continuar
... -

```

Figura 7- Entendendo a função printf – Saída / Fonte: os autores.

Ao comparar os códigos da Figura 6 e da Figura 8, podemos observar que, no segundo programa, foi inserido o código especial \n na função printf. Este código é responsável por posicionar o cursor em uma nova linha. Com isso, temos que a saída é a impressão de cada mensagem em uma linha, como pode ser visualizado na Figura 9.

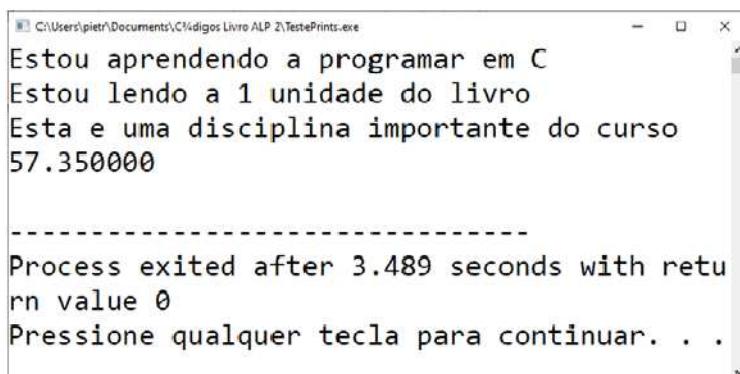


```

1 #include <stdio.h>
2 int main(){
3     printf("Estou aprendendo a programar em C\n");
4     printf("Estou lendo a %d unidade do livro\n", 1);
5     printf("%s e uma disciplina importante do curso\n", "Esta");
6     printf("%f\n", 57.35);
7     return (0);
8 }

```

Figura 8 - Entendendo a função `printf` - Código / Fonte: os autores.



```

Estou aprendendo a programar em C
Estou lendo a 1 unidade do livro
Esta e uma disciplina importante do curso
57.350000

-----
Process exited after 3.489 seconds with return value 0
Pressione qualquer tecla para continuar. . .

```

Figura 9 - Entendendo a função `printf` - Saída / Fonte: os autores.

Se você ficou com dúvidas quanto ao funcionamento do `printf`, fique tranquilo(a)! Veremos mais casos de aplicação desta função.

Construindo um programa

Vamos elaborar um programa que leia nome, idade e altura de uma pessoa e exiba nome, idade, altura e ano de nascimento dela. Para facilitar o entendimento do problema, o estruturaremos em três partes: entrada, processamento e saída.

Na entrada de dados, temos que obter os dados de nome, idade e altura. Cada um deles precisa ser armazenado em uma variável e, em sua leitura, utilizaremos a função `scanf`. Como processamento, temos que calcular o ano em que a pessoa nasceu, que será dado pelo ano atual menos a idade da pessoa. E, como saída, devemos enviar para a tela nome, idade, altura e ano de nascimento. Para mostrar estas informações no vídeo, utilizaremos a função `printf`.

Agora que entendemos o problema, podemos partir para a construção do programa. Você se recorda da estrutura básica de um programa em C? O nosso deve ter uma função `main`, e as instruções têm que estar entre as chaves que delimitam o fim e o início dessa função.

Os valores obtidos na entrada de dados precisam ser armazenados em variáveis. Qual o tipo de cada variável? O `nome` é uma cadeia de caracteres; desse modo, precisamos de uma variável vetor do tipo `char` (não se preocupe com vetores por hora, veremos, com mais detalhes, nas próximas unidades). A variável `altura` é do tipo `float` e a `idade` do tipo `int`.

O quadro a seguir apresenta o código para o programa anteriormente descrito.

```
01  /* insere o conteúdo do arquivo stdio.h */
02  #include <stdio.h>
03  int main()
04  { /* declaração das variáveis */
05      int    idade, ano;
06      float altura;
07      char   nome[30];
08      /*entrada de dados */
09      /*mensagem ao usuário */
10      printf ("Digite o seu nome: ");
11      scanf ("%s", nome); /* leitura do nome */
12      /*mensagem ao usuário */
13      printf ("Digite a idade: ");
14      scanf ("%d", &idade); /* leitura da idade */
15      /*mensagem ao usuário */
16      printf ("Digite a altura: ");
17      scanf ("%f", &altura); /* leitura da altura*/
18      /* processamento */
19      /*cálculo do ano de nascimento */
20      ano = 2012 - idade;
21      /*saída de dados */
22      printf ("\nO nome e: %s", nome);
23      printf ("\nA altura e: %f", altura);
24      printf ("\nA idade e: %d", idade);
25      printf ("\nO ano de nascimento e: %d ", ano);
26      return (0);
27 }
```

Quadro 17- Programa em C / Fonte: os autores.

Os resultados da execução do programa podem ser visualizados na Figura 10. Observe que, inicialmente, os dados foram obtidos do usuário e, em seguida, apresentados. Na saída de dados, as variáveis do tipo float podem ser formatadas em relação ao número de casas decimais a ser apresentado. Ao imprimir uma variável do tipo float, utilizando a função printf, o padrão é completar o número com zeros à direita, até que fique com seis casas decimais (Figura 8).

```
C:\Users\pietr\Documents\GitHub\Livro ALP\EntradaSaída.exe
Digite o seu nome: Ronaldo
Digite a idade: 15
Digite a altura: 1.60

O nome e: Ronaldo
A altura e: 1.600000
A idade e: 15
O ano de nascimento e: 1997
-----
Process exited after 16.23 seconds with
return value 0
Pressione qualquer tecla para continuar
... -
```

Figura 10 - Programa em C — Saída / Fonte: os autores.

Podemos formatar, no entanto, de um modo diferente, usando, junto com o especificador de formato, o número de casas decimais que desejamos. O Quadro 18 apresenta o mesmo programa com formatação para a impressão da variável altura com duas casas decimais. Na função printf, ao utilizar o especificador de formato, inserimos um ponto e o número de casas decimais desejado.



pensando juntos

A linguagem C é case sensitive. Será que existe alguma biblioteca que nos auxilie a manipular dados de textos variados, em caixa alta ou em caixa baixa?

```
01  /* insere o conteúdo do arquivo stdio.h */
02  #include <stdio.h>
03
04  int main()
05  { /* declaração das variáveis */
06      int    idade, ano;
07      float altura;
08      char   nome[30];
09      /*entrada de dados*/
10      /*mensagem ao usuário*/
11      printf ("Digite o seu nome: ");
12      scanf ("%s", nome); /* leitura do nome */
13      /*mensagem ao usuário*/
14      printf ("Digite a idade:");
15      scanf ("%d", &idade); /* leitura da idade */
16      /*mensagem ao usuário*/
17      printf ("Digite a altura:");
18      scanf ("%f", &altura); /* leitura da altura*/
19      /*processamento*/
20      /*cálculo do ano de nascimento*/
21      ano = 2012 - idade;
22      /*saída de dados*/
23      printf ("\nO nome é : %s", nome);
24      printf ("\nA altura é : %.2f", altura);
25      printf ("\nA idade é : %d", idade);
26      printf ("\nO ano de nascimento é : %d ", ano);
27
28 }
```

Quadro 18 - Programa em C / Fonte: os autores.

```
C:\Users\ptet\Documents\CT\digos Livro ALP 2\EntradaE Saída.exe
Digite o seu nome: Ronaldo
Digite a idade:15
Digite a altura: 1.623

O nome é : Ronaldo
A altura é : 1.62 ←
A idade é : 15
O ano de nascimento é : 1997
-----
Process exited after 17.48 seconds with
return value 0
Pressione qualquer tecla para continuar
***
```

Figura 11 - Programa em C — Saída formatada / Fonte: os autores.

A Figura 11 apresenta o resultado obtido com a formatação da saída para a variável real altura.



EXERCÍCIOS RESOLVIDOS

Problema 1

Escreva um programa que leia um número inteiro e apresente seu antecessor e seu sucessor.

```
01 #include <stdio.h>
02
03 int main()
04 {
05     int num, ant, suc;
06
07     printf(" Digite o número:");
08     scanf ("%d", &num);
09
10     ant = num - 1;
11     suc = num + 1;
12
13     printf("\n O antecessor é: %d", ant);
14     printf("\n O sucessor é: %d", suc);
15
16     return (0);
17 }
```

Quadro 19 - Programa em C, Problema 1 / Fonte: os autores.

```
01 #include <stdio.h>
02
03 int main()
04 {
05     int num;
06
07     printf("Digite o número: ");
08     scanf ("%d", &num);
09     printf("\n O antecessor é: %d", num-1);
10     printf("\n O sucessor é: %d", num+1);
11
12 }
```

Quadro 20 - Programa em C, Problema 1 (solução alternativa) / Fonte: os autores.

Problema 2

Elabore um programa que receba quatro notas e calcule a média aritmética entre elas.

```
01 #include <stdio.h>
02
03 int main()
04 {
05     float n1, n2, n3, n4, media;
06
07     printf("\n Digite a nota 1:");
08     scanf ("%f", &n1);
09     printf("\n Digite a nota 2:");
10     scanf ("%f", &n2);
11     printf("\n Digite a nota 3:");
12     scanf ("%f", &n3);
13     printf("\n Digite a nota 4:");
14     scanf ("%f", &n4);
15     media = (n1 + n2 + n3 + n4)/4;
16     printf("\n A média é: %.2f \n", media);
17
18 }
```

Quadro 21 - Programa em C, Problema 2 / Fonte: os autores.

Problema 3

Faça um programa que receba o valor de um depósito e o valor da taxa de juros, calcule e apresente o valor do rendimento e o valor total (valor do depósito + valor do rendimento).

```
01 #include <stdio.h>
02
03 int main()
04 {
05     float deposito, taxa, rendimento, total;
06
07     printf(" Informe o valor do depósito:");
08     scanf ("%f", &deposito);
09     printf("\n Informe a taxa de juros:");
10     scanf ("%f", &taxa);
11     rendimento = deposito * (taxa/100);
12     total = deposito + rendimento;
13     printf("\n O rendimento é: %.2f", rendimento);
14     printf("\n O total é: %.2f", total);
15
16 }
```

Quadro 22 - Programa em C, Problema 3 / Fonte: os autores.

Problema 4

Escreva um programa que receba dois números, calcule e apresente o resultado do primeiro número elevado ao segundo.

```
01 #include <stdio.h>
02 #include <math.h>
03
04 int main()
05 {
06     float num1, num2, total;
07
08     printf(" Informe o primeiro número:");
09     scanf ("%f", &num1);
10     printf("\n Informe o segundo número:");
11     scanf ("%f", &num2);
12     total = pow(num1, num2);
13     printf("\n %.2f elevado a %.2f é: %.2f", num1,
14         num2, total);
15 }
```

Quadro 23 - Programa em C, Problema 4 / Fonte: os autores.

Problema 5

Elabore um programa que calcule a área de um trapézio.

```
01 #include <stdio.h>
02
03 int main()
04 {
05     float    basel, base2, altura, area;
06
07     printf(" Informe o valor da base maior:");
08     scanf("%f", &basel);
09     printf("\n Informe o valor da base menor:");
10     scanf("%f", &base2);
11     printf("\n Informe o valor da altura:");
12     scanf("%f", &altura);
13     area = ((basel + base2) * altura)/2;
14     printf("\n A área do trapézio é: %.2f", area);
15
16 }
```

Quadro 24 - Programa em C, Problema 5 / Fonte: os autores.



CONSIDERAÇÕES FINAIS

Nesta unidade, você deu os primeiros passos no aprendizado da linguagem de programação C, conhecendo as características, os pontos fortes e fracos, e como o código que escrevemos se torna um programa executável.

Conhecemos a estrutura básica de um programa em C, que possui uma função `main`, que é o primeiro trecho de código invocado quando o programa é executado.

Vimos que o conteúdo da função é delimitado por chaves, e o conteúdo entre elas é executado de modo sequencial. Além disso, estudamos que, ao final do código-fonte, é interessante inserir o comando `return (0);`, sinalizando que o programa se encerra ali, sem erros. Além disso, vale notar que quase todos os comandos em linguagem C terminam com ponto e vírgula.

Aprendemos como documentar nossos códigos inserindo comentários, os quais são escritos entre `/* */`. Entendemos as regras para a nomeação de identificadores. Estudamos os sete tipos básicos de dados, a saber: `char`, `int`, `float`, `double`, `enum`, `void` e `pointer`.

Vimos os operadores aritméticos, relacionais e lógicos, além das funções intrínsecas disponíveis na linguagem C. Em relação aos tipos de dados, vimos que eles podem ser modificados a partir da aplicação das palavras reservadas `unsigned`, `short` e `long`.

Entendemos como realizar atribuição, entrada e saída de dados. O comando usado para atribuição é representado por `=`. A entrada de dados, normalmente, é realizada por meio da função `scanf`. E, para mostrar dados ao usuário, utilizamos a função de saída `printf`.

Nas funções de entrada e saída de dados, estudamos como utilizar os especificadores de formato e códigos especiais. Por fim, colocamos em prática a construção de um programa, utilizando os conceitos aprendidos no decorrer desta unidade.



1. Assinale V nos identificadores corretos e F nos identificadores errados, de acordo com as regras da linguagem C.

() idade
() nome*r
() media_peso%
() aluno_nota

() media idade
() x2
() endereco+cep
() A

() 2nome
() 012
() /fone
() 1234P

A sequência correta para a resposta da questão é:

- a) V, F, F, V, F, V, F, V, F, F, F.
 - b) V, F, V, V, F, V, F, V, V, V, F, F.
 - c) F, F, F, V, V, V, F, V, V, F, F, F.
 - d) V, F, F, V, F, V, F, V, F, F, V, F.
 - e) F, F, V, V, F, V, F, V, F, V, F, F.
2. Escreva um programa que leia o nome de uma pessoa e imprima a seguinte mensagem, na tela: "Bem-vindo(a) à disciplina de Algoritmos e Lógica de Programação II, <nome>", onde o campo <nome> deve ser substituído pelo nome informado pelo usuário.
3. Escreva um programa que leia um número positivo inteiro e apresente o quadrado e a raiz quadrada deste número.



4. Escreva um programa que receba quatro números inteiros, calcule e apresente a média aritmética entre eles. Observação: não esqueça de formatar o valor da média no momento de apresentá-lo, para que sejam impressas apenas duas casas decimais.
5. Escreva um programa que, dado o raio de um círculo, calcule sua área e o perímetro. A área é a superfície do objeto, dada por $A = \pi r^2$, e o perímetro é a medida do contorno do objeto dado por $P = 2\pi r^2$. Dica: utilize as funções intrínsecas vistas nesta unidade.
6. Identifique os erros sintáticos e semânticos no programa a seguir, cujo objetivo é ler dois números e apresentar a soma entre os dois.

```
01  /* insere o conteúdo do arquivo stdio.h
02  #include <stdio.h>
03
04  int main()
05  { /* declaração das variáveis */
06      int num1, num2, total;
07      /*entrada de dados */
08      /*mensagem ao usuário */
09      printf("\nDigite o primeiro número: ");
10      /* leitura do primeiro número */
11      scanf("%d", &num1);
12      /*mensagem ao usuário */
13      printf("\nDigite o segundo número:");
14      /* leitura do segundo número*/
15      scanf("%d", &num2);
16      /* processamento */
17      /*cálculo do ano de nascimento */
18      total = num1 + num2
19      /*saída de dados */
20      printf ("\n A soma dos números é : %d ", soma);
21
22 }
```



TUTORIAL DEV-C++

O Dev-C++, desenvolvido pela Bloodshed Software, é um IDE com interface gráfica (Ambiente de Desenvolvimento Integrado) completo, capaz de criar programas C/C++ para Windows ou em console usando o sistema compilador MinGW. O MinGW (Minimalist GNU* para Windows) usa o GCC (a coleção de compiladores GNU g++), que é essencialmente o mesmo sistema de compilador que está no Cygwin (o programa de ambiente UNIX para Windows) e na maioria das versões do Linux. Existem, no entanto, diferenças entre Cygwin e MinGW. Vamos ao passo a passo para poder depurar um programa.

Passo 1 – Configurando o Ambiente

Precisamos modificar uma das configurações padrão para permitir que você use o depurador (*debugger*) com seus programas.

Vá para o menu “Ferramentas” e selecione “Opções do compilador”.

Na guia “Configurações”, clique em “Linker” no painel esquerdo e altere “Gerar Informação de depuração (-g3)” para “Sim”, como ilustra a figura a seguir:

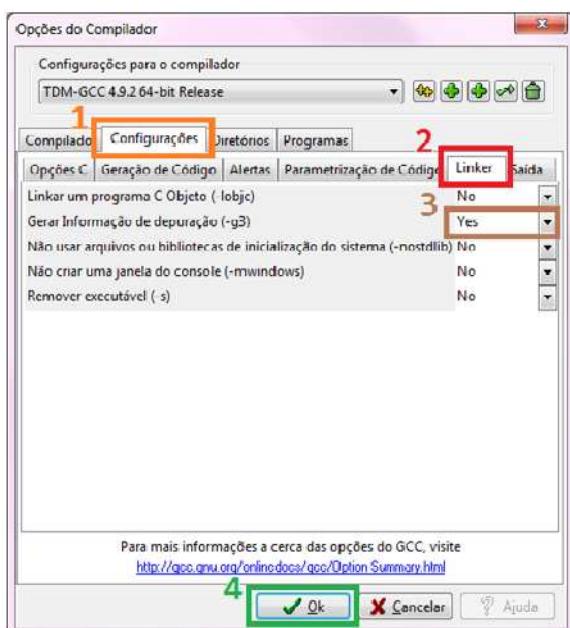


Figura 12 - Configurando o Dev-C++ para depuração.
Fonte: os autores.



Passo 2 – Criando um novo projeto

Um “projeto” pode ser considerado como um contêiner usado para armazenar todos os elementos necessários para compilar um programa. Vá para o menu “Arquivo” e selecione “Novo”, “Projeto ...”. Escolha “Projeto vazio” e verifique se “Projeto C” está selecionado. Aqui você também dará um nome ao seu projeto. Você pode atribuir um nome de arquivo válido ao seu projeto, mas lembre-se de que o nome do seu projeto também será o nome do seu executável final. Depois de inserir um nome para o seu projeto, clique em “OK”. Agora, o Dev-C ++ perguntará onde salvar seu projeto.

Passo 3 – Criando/adicionando arquivos fonte

Você pode adicionar arquivos fonte vazios de duas maneiras distintas:

- I - Vá para o menu “Arquivo” e selecione “Novo > Arquivo Fonte” (ou apenas pressione CTRL + N) OU;
- II - Vá para o menu “Projeto” e selecione “Novo arquivo”.

Observe que o Dev-C ++ não solicitará um nome de arquivo para nenhum novo arquivo fonte até que você tente realizar alguma dentre as quatro ações: tente compilar; salve o projeto; salve o arquivo de origem; saia do Dev-C++.

Você pode adicionar arquivos de origem preexistentes de uma das duas maneiras:

- I - Vá para o menu “Projeto” e selecione “Adicionar ao projeto” OU;
- II - Clique com o botão direito do mouse no nome do projeto no painel esquerdo e selecione “Adicionar ao projeto”.



Passo 4 – Compilar

Depois de inserir todo o seu código-fonte, você estará pronto para compilar. Vá para o menu “Executar” e selecione “Compilar” (ou apenas pressione F9). É provável que você receba algum tipo de erro de compilador ou linker na primeira vez que tentar compilar um projeto. Os erros de sintaxe serão exibidos na guia “Compilador” na parte inferior da tela. Você pode clicar duas vezes em qualquer erro para levá-lo ao local no código-fonte em que ocorreu. A guia “Linker” piscará se houver algum erro no linker. Erros de linker geralmente são o resultado de erros de sintaxe que não permitem a compilação de um dos arquivos. Depois que seu projeto for compilado com êxito, a caixa de diálogo “Registro do Compilador” mostrará o nome do arquivo executável produzido, o tamanho desse arquivo e o tempo total para compilar o arquivo.

Passo 5 – Executar

Agora você pode executar seu programa. Vá para o menu “Executar”, escolha “Executar”. Nota: para passar parâmetros de linha de comando para o seu programa, vá para o menu “Executar”, escolha “Parâmetros” e digite os parâmetros que deseja passar.

Se você executar seu programa (com ou sem parâmetros), poderá notar algo peculiar – uma janela do console poderá aparecer, piscar algum texto e desaparecer. O problema é que, se executadas diretamente, as janelas do programa do console fecham após a saída do programa. Você pode resolver esse problema de duas maneiras:

- Método 1 - Adicionando uma chamada de sistema:

Na linha antes do retorno da função `main`, insira: `system("pause");`



- Método 2 - *prompt* de comando:

Como alternativa, em vez de usar o Dev-C ++ para chamar seu programa, você pode simplesmente abrir um *prompt* do MS-DOS, ir para o diretório em que seu programa foi compilado (ou seja, onde você salvou o projeto) e inserir o nome do programa (junto com qualquer parâmetro). A janela do *prompt* de comando não fechará quando o programa terminar.

Passo 6 – Depuração (*debug*)

Quando as coisas não estão acontecendo como planejado, um depurador pode ser uma ótima ferramenta para determinar o que realmente está acontecendo durante a execução do seu programa. As funções básicas do depurador do Dev-C++ são controladas por meio da guia “Depurador” na parte inferior da tela; funções mais avançadas estão disponíveis no menu “Depurador”.

Para iniciar a depuração, esteja certo de que o programa esteja compilado de que não está em execução e, em seguida, acesse “Executar” e depois “Depurar” (ou simplesmente pressione F5). É possível adicionar pontos de parada (*breakpoints*) no seu código-fonte; basta clicar no número da linha na qual você deseja pausar a execução durante o processo de depuração. Os vários recursos do depurador são bastante óbvios. Clique em “Próxima Instrução” para percorrer o código; clique em “Adicionar Marcador” para monitorar variáveis.

Fonte: adaptado de Niño ([2020], on-line)¹.



eu recomendo!



livro

Programação em Linguagem C

Autor: R. S. Albano

Editora: Ciência Moderna

Ano: 2010

Sinopse: a linguagem C é utilizada na área de programação. O livro *Programação em linguagem C* oferece mais de 200 códigos-fontes, distribuídos entre exemplos e exercícios de fixação. É indicado para alunos de cursos de graduação, cursos técnicos ou livres. Além disso, os autodidatas poderão utilizá-lo, já que abrange, de forma sequencial, a fase introdutória da linguagem de programação C até a sua fase intermediária. Esta obra contém vários exercícios executados passo a passo que permitem que o leitor possa acompanhar o desenvolvimento de maneira útil e eficaz. Desta forma, o próprio leitor poderá implementar cada exercício à medida que lê. Apresenta-se estruturado de forma que, sempre ao final de cada capítulo, sejam apresentados exercícios de revisão abrangendo cada conteúdo estudado, com o objetivo de avaliar e consolidar os conhecimentos adquiridos. Salienta-se que todos os exercícios possuem resolução ao final.





ESTRUTURA CONDICIONAL

PROFESSORES

Dra. Gislaine Camila Lapasini Leal
Me. Pietro Martins de Oliveira

PLANO DE ESTUDO

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Estrutura condicional
- Estrutura condicional simples
- Estrutura condicional composta
- Estrutura case
- Exercícios resolvidos

OBJETIVOS DE APRENDIZAGEM

- Compreender a estrutura condicional
- Conhecer a estrutura condicional simples
- Conhecer a estrutura condicional composta
- Conhecer a estrutura case
- Elaborar algoritmos utilizando estrutura condicional.

INTRODUÇÃO



Nesta unidade, você estudará as estruturas de decisão, também conhecidas como estruturas de seleção. Estas podem ser categorizadas em estrutura condicional simples, condicional composta e estrutura `case` (estrutura condicional múltipla). Com os conhecimentos adquiridos na Unidade 1, conseguimos construir programas sequenciais, isto é, em que, a partir da entrada, os dados são processados sequencialmente, sem desvios de fluxo. Ao fim do algoritmo, apresentamos algumas informações na saída.

Na estrutura condicional, podemos impor condições para a execução de uma instrução ou um conjunto de instruções, isto é, podemos criar condições que permitem desviar o fluxo de execução de um programa. Para construir estas condições, utilizaremos os conceitos de variáveis, de expressões lógicas e expressões relacionais, vistos na Unidade 1.

Estudaremos a estrutura condicional simples, que nos permite tomar uma decisão entre executar ou não um bloco de comandos. Veremos, também, a estrutura condicional composta que, a partir de uma expressão, pode definir dois caminhos distintos: um quando o resultado do teste é `verdadeiro`, e outro, quando o resultado é `falso`. E a estrutura `case`, que é uma generalização das estruturas condicionais em que pode haver uma ou mais condições de igualdade a serem testadas e cada uma delas pode ter uma instrução diferente associada. Além disso, por meio da combinação entre operadores lógicos e relacionais, é possível compor condições complexas dentro de uma mesma estrutura de seleção.

Ao estudar cada estrutura condicional, veremos que, em alguns exemplos, teremos condições de construir algoritmos para visualizar a aplicação dos conceitos estudados. No final desta unidade, estaremos aptos a construir programas com desvio de fluxo, aumentando, assim, o leque de problemas que podemos solucionar.



1 ESTRUTURA CONDICIONAL

A estrutura condicional é fundamental para qualquer linguagem de programação, uma vez que, sem ela, o fluxo de execução dos programas seria seguido sequencialmente, sem nenhum desvio, ou seja, instrução a instrução. A estrutura condicional possibilita o desvio do fluxo do programa, sendo, também, denominada estrutura de seleção ou estrutura de controle (MANZANO; OLIVEIRA, 1997; ASCENCIO; CAMPOS, 2010).

A estrutura condicional consiste em uma estrutura de controle de fluxo que permite executar um ou mais comandos se a condição testada for verdadeira, ou executar um ou mais comandos, se for falsa (LOPES; GARCIA, 2002).

Nas aulas seguintes, estudaremos as estruturas condicionais da linguagem C.



Para saber um pouco mais sobre a estrutura condicional em C, acesse:
http://pt.wikibooks.org/wiki/Programar_em_C/Estudo#Branching_-_IF.

Fonte: os autores.





2 ESTRUTURA CONDICIONAL simples

Na estrutura condicional simples, o bloco de comandos só será executado se a condição for verdadeira. Uma condição é uma comparação que possui dois valores possíveis: verdadeiro ou falso (ASCENCIO; CAMPOS, 2010).

A sintaxe do comando é:

```
if (<condição>)
{
    <bloco de comandos p/ condição verdadeira>;
}
```

A estrutura condicional simples tem por finalidade tomar uma decisão. De modo que, se a condição que está sendo testada for verdadeira, são executadas todas as instruções compreendidas entre { }. Ao término da execução destas instruções, o algoritmo segue o primeiro comando após “ } ”. Por outro lado, se a condição que está sendo testada for falsa, o algoritmo executa a primeira instrução após o “ } ”, não executando as instruções compreendidas entre { }.

Em C, é obrigatório o uso de chaves quando existe mais de um comando a executar. As chaves delimitam um novo bloco de comandos. Além disso, não podemos esquecer de inserir o “ ; ” ao final dos comandos que o exigem.

Agora que conhecemos a sintaxe da estrutura condicional simples, construiremos nosso primeiro programa em C com desvio de fluxo. O problema consiste em obter um número inteiro e, se este for par, imprimir sua raiz quadrada.

O Quadro 1 apresenta o programa em C para o problema descrito anterior. Lembre-se de que os textos compreendidos entre `/* */` correspondem a comentários e não são executados como comandos.

Note que, neste programa, inserimos o conteúdo da biblioteca `stdio.h` e `math.h`. A biblioteca `math.h` é própria para cálculos matemáticos e inclui funções, tais como: potência, raiz quadrada, funções trigonométricas e outras.

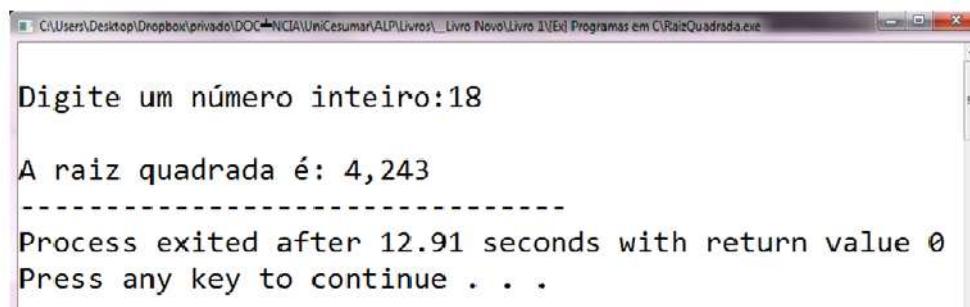
Em relação às variáveis, foram declaradas duas delas: `num` e `raiz`. O número inteiro obtido do usuário é armazenado na variável `num`, e o resultado do cálculo da raiz quadrada deste número é armazenado na variável `raiz`. O teste lógico, realizado na estrutura condicional simples (`if` - linha 16), consiste em verificar se o valor do resto da divisão do número lido por dois é igual a zero. Se esta condição for verdadeira, é executado o conjunto de instruções delimitado pelas chaves (“`{ }`”), isto é, temos a execução da instrução de atribuição do valor da raiz quadrada do número à variável `raiz`, seguido da impressão de uma mensagem ao usuário informando o resultado. Se o valor do teste lógico (linha 16) for falso, é executada a linha com o comando `return (0)` (linha 21).

```
01  /* Estrutura condicional simples em C */
02  /*incluindo o conteúdo de stdio.h */
03  #include <stdio.h>
04  /*insere o conteúdo do arquivo math, que tem a função sqrt, cálculo da raiz quadrada */
05  #include <math.h>
06
07
08  /* todo programa em C deve possuir essa linha */
09  int main()
10  { /* delimita o início das instruções */
11      int num;
12      float raiz;
13
14      printf("\nDigite um número inteiro:");
15      scanf ("%d", &num);
16      if (num % 2 == 0 )
17      {
18          raiz = sqrt(num);
```

```
19         printf ("\nA raiz quadrada é: %.3f", raiz);
20     }
21     return (0); /* retorno sem erro */
22 } /*fim do conjunto de instruções da função main */
```

Quadro 1 - Programa em C / Fonte: os autores

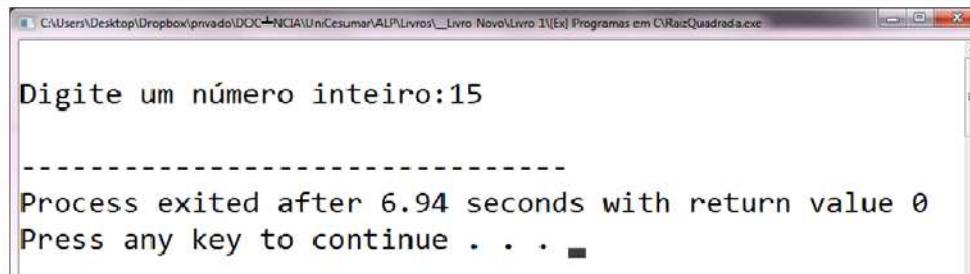
Analisaremos o resultado da simulação do programa para os valores 18 (Figura 1) e 15 (Figura 2), respectivamente. Como 18 é um número par, o resultado do teste lógico da linha 16 é verdadeiro. Com isso, temos a impressão do valor da raiz quadrada, no caso, 4.243 (execução das linhas 18 e 19). Observe que o valor apresentado possui três casas decimais devido à formatação `%.3f`, definida no comando `printf()` da linha 19.



```
C:\Users\Desktop\Dropbox\privado\DOC-NCIA\Unicesar\ALP\Livros\...\\Ex] Programas em C\RaizQuadrada.exe

Digite um número inteiro:18
A raiz quadrada é: 4,243
-----
Process exited after 12.91 seconds with return value 0
Press any key to continue . . .
```

Figura 1 - Programa em C – Saída / Fonte: os autores.



```
C:\Users\Desktop\Dropbox\privado\DOC-NCIA\Unicesar\ALP\Livros\...\\Ex] Programas em C\RaizQuadrada.exe

Digite um número inteiro:15
-----
Process exited after 6.94 seconds with return value 0
Press any key to continue . . .
```

Figura 2 - Programa em C – Saída / Fonte: os autores.

Observe a Figura 2, no caso do valor 15, o resultado do teste lógico é falso. Deste modo, as instruções compreendidas pelo bloco de comandos entre as chaves, das linhas 17 a 20, não são executadas, portanto, temos a execução da instrução da linha 21, o `return (0);`.

ESTRUTURA CONDICIONAL composta

Na estrutura condicional composta, é realizada a avaliação de uma única expressão lógico-relacional. Se o resultado desta avaliação for verdadeiro, é executada a instrução ou o conjunto de instruções compreendidas entre as chaves após o **if**. Caso contrário, se o resultado da avaliação for falso, é executada a instrução ou o conjunto de instruções entre chaves após o **else**.

A sintaxe da estrutura condicional composta é:

```
if (<condição>)
{
    <Instruções para condição verdadeira>;
}
else
{
    <Instruções para condição falsa>;
}
```

Para facilitar o entendimento quanto ao funcionamento da estrutura condicional composta, construiremos um programa para identificar se um número é par ou ímpar. Se o número for par, devemos apresentar sua raiz quadrada e, se for ímpar, devemos apresentar o número elevado ao quadrado.

No quadro a seguir, é apresentado um programa em C que verifica se o número é par ou ímpar. Em relação às variáveis, foram declaradas três: num, quadrado e raiz. O número inteiro obtido do usuário é armazenado na variável num.

No teste lógico realizado na estrutura condicional da linha 12, verifica-se que o valor do resto da divisão do número lido por dois é igual a zero. Se esta condição for verdadeira, é executado o conjunto de instruções do **if**, delimitado pelas chaves que vão da linha 13 à linha 17, isto é, temos a execução da instrução de atribuição do valor da raiz quadrada do número à variável raiz (linha 14), seguido da impressão de uma mensagem ao usuário informando o resultado (linhas 15 e 16). Se o valor do teste lógico da linha 12 for **false**, é executado o conjunto de instruções do **else**, delimitado pelas chaves que vão da linha 19 à linha 23, em que é calculado o valor de num ao quadrado.

```
01  /* Estrutura condicional composta em C */
02  #include <stdio.h>
03  #include <math.h>
04
05  int main()
06  {
07      int quadrado, num;
08      float raiz;
09
10      printf("\nDigite um número inteiro:");
11      scanf ("%d", &num);
12      if (num % 2 == 0)
13      {
14          raiz = sqrt(num);
15          printf ("\nO número é par.");
16          printf ("\nA raiz quadrada é: %.3f", raiz);
17      }
18      else
19      {
20          quadrado = num * num;
21          printf ("\nO número é ímpar.");
22          printf ("\nO número ao quadrado é: %d", quadrado);
23      }
24  }
25 }
```

Quadro 2 - Programa em C / Fonte: os autores.

Tomemos como exemplo os valores 15 (Figura 3) e 16 (Figura 4), respectivamente. Analisaremos o resultado da execução do programa para o valor 15. Na avaliação do teste lógico da linha 12, temos que o resto da divisão de 15 por dois não é igual a zero, isto é, o resultado do teste lógico é falso. Com isto, temos a execução do conjunto de instruções após o **else**. O resultado da execução é apresentado na figura a seguir.

The screenshot shows a terminal window with the following text:
Digitie um número inteiro:15
O número é ímpar.
O número ao quadrado é: 225

Process exited after 7.028 seconds with return value 0
Press any key to continue . . .

Figura 3 - Programa em C – Saída / Fonte: os autores.

The screenshot shows a terminal window with the following text:
Digitie um número inteiro:16
O número é par.
A raiz quadrada é: 4,000

Process exited after 6.802 seconds with return value 0
Press any key to continue . . .

Figura 4 - Programa em C – Saída / Fonte: os autores.

Ao analisar a execução que insere o número 16, temos que o resultado do teste lógico é verdadeiro. Deste modo, é executado o conjunto de instruções após o **if**, como pode ser visto na Figura 4.



4 ESTRUTURA CASE



A estrutura **case** consiste em uma generalização do **if**, em que somente uma condição era avaliada e dois caminhos poderiam ser seguidos: um para o resultado da avaliação ser verdadeiro, e outro, para falso. Na estrutura case, pode haver uma ou mais condições a serem avaliadas e um comando diferente associado a cada uma delas, isto é, uma construção de múltiplas possibilidades de decisão, que compara o resultado de uma expressão com uma série de valores constantes (LOPES; GARCIA, 2002).

A estrutura **case** é utilizada quando temos situações mutuamente exclusivas, em que, se uma instrução for executada, as demais não serão. Para este tipo de situação, recomenda-se o uso de um comando seletivo (ASCENCIO; MOURA, 2010).

A sintaxe desta estrutura é:

```
switch (<variável>)
{
    case <valor 1>: <instruções>;
    break;
    case <valor 2>: <instruções>;
    break;
    case <valor 3>: <instruções>;
    break;
```

```
    default: <instruções>;  
    break;  
}
```

Nessa estrutura, o comando **switch** avalia o valor da variável para decidir qual **case** será executado. Cada **case** representa um possível valor da variável, o qual, obrigatoriamente, deve ser do tipo **char**, **unsigned char**, **int**, **unsigned int**, **long** ou **unsigned long**. Para impedir a execução das instruções definidas nos demais **case**, deve-se utilizar o comando **break**. Se o valor da variável não coincidir com aqueles especificados nos **case**, são executadas as instruções do **default** (ASCENCIO; CAMPOS, 2010).

Lopes e Garcia (2002) destacam que a estrutura **case** é bastante utilizada na construção de menus, tornando-os mais claros. Para ilustrar o funcionamento da estrutura **case**, construiremos um programa que permita ao usuário escolher qual operação deseja realizar com dois números (1 – soma, 2 – subtração, 3 – multiplicação e 4 – divisão). Precisamos de duas variáveis para armazenar cada um dos números: uma para armazenar a operação selecionada pelo usuário, e outra para armazenar o resultado da operação. Deste modo, temos a declaração de três variáveis do tipo **float** (**num1**, **num2** e **resultado**) e uma variável do tipo **int** (**op**).

A entrada de dados consiste na leitura dos dois números e da operação desejada. Como processamento, temos que identificar a operação selecionada, utilizando a estrutura **case**, e efetuá-la. A saída de dados é a impressão do resultado da operação. O programa em C é apresentado no quadro a seguir:



```
01  /* Estrutura case em C */
02  #include <stdio.h>
03
04  int main()
05  {
06      float num1, num2, resultado;
07      int op;
08
09      printf("Digite o primeiro número:\n");
10      scanf("%f", &num1);
11      printf("Digite o segundo número:\n");
12      scanf("%f", &num2);
13      printf("Escolha a operação: \n 1 - Soma \n 2 -\n Subtração \n 3 - Multiplicação \n 4 - Divisão \n");
14      scanf("%d", &op);
15      switch (op)
16      {
17          case 1:
18              resultado = num1 + num2;
19              printf("A soma é: %.3f\n", resultado);
20              break;
21          case 2:
22              resultado = num1 - num2;
23              printf("A subtração é: %.3f\n", resultado);
24              break;
25          case 3:
26              resultado = num1 * num2;
27              printf("A multiplicação é: %.3f\n",
28                     resultado);
29              break;
30          case 4:
31              resultado = num1 / num2;
32              printf("A divisão é: %.3f\n", resultado);
33              break;
34          default:
35              printf("Opção inválida.\n");
36              break;
37      }
38  }
```

Quadro 3 - Programa em C / Fonte: os autores

O resultado obtido na execução do programa para cada uma das operações é ilustrado nas Figuras 5, 6, 7 e 8.

```
C:\Users\Desktop\Dropbox\privado\DOC-4-NCIA\Unicesumar\ALP\Livro\_\Livro Novo\Livro 1\Ex Programas em C\Calculadora.exe  
Digite o primeiro número:  
10  
Digite o segundo número:  
15  
Escolha a operação:  
1 - Soma  
2 - Subtração  
3 - Multiplicação  
4 - Divisão  
1  
A soma é: 25,000  
-----  
Process exited after 10.52 seconds with return value 0  
Press any key to continue . . .
```

Figura 5 - Programa em C – Saída / Fonte: os autores.

```
C:\Users\Desktop\Dropbox\privado\DOC-4-NCIA\Unicesumar\ALP\Livro\_\Livro Novo\Livro 1\Ex Programas em C\Calculadora.exe  
Digite o primeiro número:  
15  
Digite o segundo número:  
10  
Escolha a operação:  
1 - Soma  
2 - Subtração  
3 - Multiplicação  
4 - Divisão  
2  
A subtração é: 5,000  
-----  
Process exited after 8.636 seconds with return value 0  
Press any key to continue . . .
```

Figura 6 - Programa em C – Saída / Fonte: os autores.

```
C:\Users\Desktop\Dropbox\privado\DOC-4-NCIA\UniCesumar\ALP\Livros\..._Livre Novo\Livre 1\Ex] Programas em C\Calculadora.exe
Digite o primeiro número:
10
Digite o segundo número:
15
Escolha a operação:
1 - Soma
2 - Subtração
3 - Multiplicação
4 - Divisão
3
A multiplicação é: 150,000

-----
Process exited after 8.991 seconds with return value 0
Press any key to continue . . .
```

Figura 7 - Programa em C – Saída / Fonte: os autores.

```
C:\Users\Desktop\Dropbox\privado\DOC-4-NCIA\UniCesumar\ALP\Livros\..._Livre Novo\Livre 1\Ex] Programas em C\Calculadora.exe
Digite o primeiro número:
15
Digite o segundo número:
10
Escolha a operação:
1 - Soma
2 - Subtração
3 - Multiplicação
4 - Divisão
4
A divisão é: 1,500

-----
Process exited after 8.849 seconds with return value 0
Press any key to continue . . .
```

Figura 8 - Programa em C – Saída / Fonte: os autores.



Para compreender se seus programas estão funcionando como esperado, é preciso realizar vários testes, executando o mesmo programa várias vezes seguidas. A cada execução, porém, é interessante inserir valores distintos para tentar encontrar falhas.

Fonte: os autores.

Você já se perguntou como ficaria um programa para o mesmo problema sem utilizar a estrutura `case`? Podemos resolver de outra forma? O problema pode ser resolvido utilizando a estrutura condicional composta, como pode ser visto no quadro a seguir.

```
01 #include <stdio.h>
02
03 int main()
04 {
05     float num1, num2, resultado;
06     int op;
07
08     printf("Digite o primeiro número:\n");
09     scanf("%f", &num1);
10     printf("Digite o segundo número:\n");
11     scanf("%f", &num2);
12     printf("Escolha a operação: \n 1 - Soma \n 2 -\n Subtração \n 3 - Multiplicação \n 4 - Divisão \n");
13     scanf("%d", &op);
14     if (op == 1) {
15         resultado = num1 + num2;
16         printf ("A soma é: %.3f\n", resultado);
17     }
18     else {
19         if (op == 2) {
20             resultado = num1 - num2;
21             printf ("A subtração é: %.3f\n",
22                     resultado);
23         }
24         else {
25             if (op == 3) {
26                 resultado = num1 * num2;
27                 printf ("A multiplicação é: %.3f\n",
28                         resultado);
29             }
30             else
31             {
32                 resultado = num1 / num2;
33                 printf ("A divisão é: %.3f\n",
34                         resultado);
35             }
36     }
37 }
```

Quadro 4 - Programa em C / Fonte: os autores.

Neste ponto, você pode se questionar quando utilizar cada uma das estruturas. Para auxiliar a compreensão das estruturas condicionais, são apresentadas algumas diretrizes no quadro a seguir:

if	Quando um bloco deve ser executado (apenas se uma condição for verdadeira).
if-else	Quando uma condição implica a execução de um ou outro bloco. Em situações em que há duas condições mutuamente exclusivas.
switch-case	Para testar uma expressão que gere valores discretos. Quando o número de possibilidades de escolha for razoavelmente grande e finito.

Quadro 5 - Estruturas condicionais / Fonte: os autores.

O Quadro 6 apresenta o algoritmo para o problema de receber dois números e escolher a operação aritmética desejada, utilizando a estrutura condicional simples. Note que não há uma regra que nos diga qual estrutura utilizar. Em muitos casos, podemos utilizar qualquer uma.

Neste ponto, você deve estar se perguntando qual a diferença entre os programas apresentados no Quadro 3, no Quadro 4 e no Quadro 6. Os três programas são soluções para o problema descrito, o que os diferencia é a eficiência. Esta pode ser analisada em função do número de operações, no caso, comparações que o programa deve realizar.

```

01 include <stdio.h>
02
03 int main()
04 {
05     float num1, num2, resultado;
06     int    op;
07
08     printf("Digite o primeiro número:\n");
09     scanf("%f", &num1);
10     printf("Digite o segundo número:\n");
11     scanf("%f", &num2);
12     printf("Escolha a operação: \n 1 - Soma \n 2 -
Subtração \n 3 - Multiplicação \n 4 - Divisão
\n");

```

```
13     scanf("%d", &op);
14     if (op == 1) {
15         resultado = num1 + num2;
16         printf("A soma é: %.3f\n", resultado);
17     }
18     if (op == 2) {
19         resultado = num1 - num2;
20         printf("A subtração é: %.3f\n", resultado);
21     }
22     if (op == 3) {
23         resultado = num1 * num2;
24         printf("A multiplicação é: %.3f\n",
25                resultado);
26     }
27     if (op == 4) {
28         resultado = num1 / num2;
29         printf("A divisão é: %.3f\n", resultado);
30     }
31 }
```

Quadro 6 - Programa em C / Fonte: os autores.

Observe que, ao utilizar a estrutura condicional simples (Quadro 6), todos os testes lógicos serão realizados. Por exemplo, se a opção informada for a soma, temos que o valor da variável `op` é 1. A primeira condição será testada (linha 14) e o resultado será `verdadeiro` e as instruções compreendidas entre as linhas 14 e 17 serão executadas. A partir daí, os demais testes lógicos das linhas 18, 22 e 26 também serão avaliados, e as instruções não serão executadas porque o resultado será `falso`. Ao utilizar a estrutura condicional composta ou a estrutura `case`, temos menor número de testes lógicos para ser avaliado.

Ao analisar o código da estrutura condicional composta e da estrutura `case`, podemos notar que é mais fácil compreender o funcionamento do programa utilizando a `case`, pois o código é mais claro. Fique tranquilo(a)! Com a realização das atividades práticas, você conseguirá definir com mais facilidade quando é melhor utilizar cada uma das estruturas.



EXERCÍCIOS RESOLVIDOS

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

5

Problema 2

Elabore um programa que receba o nome e a idade de uma pessoa e informe o nome, a idade e o valor da mensalidade do plano de saúde. A tabela a seguir apresenta os valores de mensalidade:

Até 18 anos	R\$ 50,00
De 19 a 29 anos	R\$ 70,00
De 30 a 45 anos	R\$ 90,00
De 46 a 65 anos	R\$ 130,00
Acima de 65 anos	R\$ 170,00

Tabela 1 - Valor de mensalidade de acordo com a faixa etária

Fonte: os autores.

```
01 #include <stdio.h>
02 int main()
03 {
04     char nome[30];
05     int idade;
06
07     printf("Informe o nome:\n");
08     scanf ("%s", nome);
09     printf("Informe a idade:\n");
10     scanf ("%d", &idade);
11
12     printf ("Nome: %s\n", nome);
13     printf ("Idade: %d\n", idade);
14     if (idade <= 18) {
15         printf("O valor do plano é: R$50,00\n");
```

```
16 }
17     else
18     {
19         if ((idade >=19) && (idade <= 29))
20         {
21             printf("O valor do plano é: R$70,00\n");
22         }
23     else
24     {
25         if ((idade >=30) && (idade <= 45))
26         {
27             printf("O valor do plano é: R$90,00\n");
28         }
29     else
30     {
31         if ((idade >= 46) && (idade <= 65))
32         {
33             printf("O valor do plano é:
34                 R$130,00\n");
35         }
36     else
37     {
38         printf("O valor do plano é:
39                 R$170,00\n");
40     }
41 }
42 return (0);
43 }
```

Quadro 8 - Programa em C, Problema 2 / Fonte: os autores.

Problema 3

Construa um programa que receba a idade de uma pessoa e identifique sua classe eleitoral: não eleitor (menor que 16 anos de idade), eleitor obrigatório (entre 18 e 65 anos) e eleitor facultativo (entre 16 e 18 anos e maior que 65 anos).

```
01 #include <stdio.h>
02 int main()
03 {
04     int idade;
05     printf("Informe a idade:\n");
06     scanf("%d", &idade);
07     if (idade < 16)
08     {
09         printf ("Não eleitor\n");
10     }
11     else
12     {
13         if ((idade < 18) || (idade > 65))
14         {
15             printf ("Eleitor facultativo\n");
16         }
17         else
18         {
19             printf ("Eleitor obrigatório\n");
20         }
21     }
22     return (0);
23 }
```

Quadro 9 - Programa em C, Problema 3 / Fonte: os autores.

Problema 4

De acordo com uma tabela médica, o peso ideal está relacionado com a altura e o sexo. Elabore um algoritmo que receba altura e sexo de uma pessoa e calcule e imprima o seu peso ideal, sabendo que:

Para homens	$(72.7 \times \text{altura}) - 58$
Para mulheres	$(62.1 \times \text{altura}) - 44.7$

```

01 #include <stdio.h>
02 int main()
03 {
04     float altura, peso;
05     char sexo;
06
07     printf("Informe o sexo (M/F):\n");
08     scanf("%c", &sexo);
09     printf("Informe a altura:\n");
10     scanf ("%f", &altura);
11     if ((sexo=='F') || (sexo=='f'))
12     {
13         peso = (62.1* altura) - 44.7;
14     }
15     else
16     {
17         peso = (72.7*altura) - 58;
18     }
19     printf ("O sexo é: %c\n", sexo);
20     printf ("A altura é: %.2f\n", altura);
21     printf ("O peso ideal é: %.2f\n", peso);
22     return (0);
23 }
```

Quadro 10 - Programa em C, Problema 4 / Fonte: os autores.



A estrutura `case` só pode ser utilizada em situações mutuamente exclusivas. Poderíamos, todavia, utilizar as outras estruturas de decisão, como `if` e `else` para substituir a estrutura `case`?

Problema 5

Faça um programa que informe a quantidade total de calorias a partir da escolha do usuário, que deve informar o prato típico e a bebida. A tabela de calorias encontra-se a seguir:

Prato	Bebida
Italiano 750 cal	Chá 30 cal
Japonês 324 cal	Suco de laranja 80 cal
Salvadorenho 545 cal	Refrigerante 90 cal

Tabela 2 - Tabela de calorias / Fonte: os autores.

```
01 #include <stdio.h>
02 int main()
03 {
04     int op;
05     float total;
06
07     total = 0;
08     printf("1 - Italiano 2 - Japonês 3 - Salvadorenho
09 \n");
10     printf("Informe o prato desejado: \n");
11     scanf("%d", &op);
12     switch (op)
13     {
14         case 1: total = total + 750;
15             break;
16         case 2: total = total + 324;
17             break;
18         case 3: total = total + 545;
19             break;
20     }
21     printf("1 - Chá 2 - Suco de laranja 3 -
22 Refrigerante: \n");
23     printf("Informe a bebida desejada: \n");
24     scanf("%d", &op);
25     switch (op)
26     {
27         case 1: total = total + 30;
28             break;
29         case 2: total = total + 80;
30             break;
31         case 3: total = total + 90;
32             break;
33     }
34     printf ("O total de calorias é: %.2f \n", total);
35     return (0);
36 }
```

Quadro 11 - Programa em C, Problema 5 / Fonte: os autores.

CONSIDERAÇÕES FINAIS

Com essas estruturas condicionais, é possível elaborar programas em que a execução de uma instrução ou um conjunto de instruções está atrelada(o) à avaliação de um teste lógico.

Estudamos as estruturas condicionais simples, compostas e a estrutura **case**. Vimos que o uso da estrutura condicional simples é realizado em situações em que um conjunto de instruções deve ser executado apenas se uma condição for verdadeira.

Aprendemos que, na estrutura condicional composta, é realizada a avaliação de uma única expressão lógica, no entanto, temos dois caminhos para seguir, dos quais sempre teremos um dos dois caminhos sendo executado. Se o resultado dessa avaliação for **verdadeiro**, é executada a instrução ou o conjunto de instruções compreendidas entre as chaves após o **if**. Se o resultado da avaliação for **falso**, é executada a instrução ou o conjunto de instruções entre as chaves após o **else**.

Conhecemos a estrutura **case**, que compara o resultado de uma expressão com uma série de valores constantes, em que pode haver uma ou mais condições a serem avaliadas e um comando diferente associado a cada uma delas. Vimos que esta estrutura deve ser utilizada em situações mutuamente exclusivas.

Entendemos que a diferença associada ao uso de cada uma dessas estruturas está relacionada à eficiência do programa, a qual pode ser avaliada em função do número de operações realizadas. Por fim, construímos um programa utilizando a estrutura condicional e trabalhamos a construção de expressões.

Ao longo desta unidade, construímos programas utilizando as funções de entrada e saída de dados vistas na Unidade 1 e colocamos em prática o uso das estruturas condicionais. Se você ficou com alguma dúvida, analise os exercícios de fixação, mas lembre-se: para que você fixe os conceitos, é importante que faça as atividades de **autoestudo**.



1. É comum, em uma aplicação, ter de determinar quais números são o maior ou o menor, dentre todos os valores de um conjunto de dados. Assim sendo, escreva um programa que receba cinco números inteiros e apresente o maior e o menor.
2. Em algumas situações, em uma aplicação, é preciso determinar quais são os números múltiplos de um ou mais valores, dentre todos os valores de um conjunto de dados. Dessa forma, faça um programa que leia um número e informe se ele é divisível por três e por sete.
3. Considere uma aplicação que necessita de um calendário embutido. Assim sendo, construa um programa que, dado um número inteiro, informe o mês correspondente.
4. Elabore um programa que receba o salário de um funcionário e o código do cargo e apresente o cargo, o valor do aumento e o novo salário. A tabela a seguir apresenta os cargos.

Código	Cargo	Percentual do aumento
1	Servente	40%
2	Pedreiro	35%
3	Mestre de obras	20%
4	Técnico de segurança	10%



5. Faça um programa que receba o código do estado de origem da carga de um caminhão, o peso da carga em toneladas e o código dela.

Código	Cargo
1	20
2	15
3	10
4	5

Código da carga	Preço por quilo
10 a 20	180
21 a 30	120
31 a 40	230

Calcule e apresente: o peso da carga em quilos, o preço da carga, o valor do imposto e o valor total da carga (preço da carga mais imposto).



INTELIGÊNCIA ARTIFICIAL: SISTEMAS ESPECIALISTAS NO GERENCIAMENTO DA INFORMAÇÃO

As estruturas de seleção são a base para a criação de sistemas que podem auxiliar os seres humanos no processo de tomada de decisão. Existe uma classe especial de sistemas que têm esta característica: são os ditos Sistemas Especialistas. No artigo a seguir, você conhecerá um pouco mais deste tipo de sistema.

A expressão inteligência artificial está associada, geralmente, ao desenvolvimento de sistemas especialistas. Estes sistemas, baseados em conhecimento, construídos, principalmente, com regras que reproduzem o conhecimento do perito, são utilizados para solucionar determinados problemas em domínios específicos. A área médica, desde o início das pesquisas, tem sido uma das áreas mais beneficiadas pelos sistemas especialistas, por ser considerada detentora de problemas clássicos com todas as peculiaridades necessárias para serem instrumentalizados por tais sistemas.

Nem todos os problemas devem ser resolvidos por meio de sistemas especialistas. Existem características que indicam se determinado problema deve ou não ser instrumentalizado por esta tecnologia. A análise do problema, então, constitui-se no primeiro estágio do ciclo de desenvolvimento dos sistemas especialistas, contribuindo fortemente para o sucesso da implementação do sistema. Buscando facilitar o processo de análise do problema, distinguimos, dentre outras, algumas condições, que, se observadas, poderão contribuir para a identificação do nível de adequação do uso da tecnologia de sistemas especialistas para a sua resolução:

- **existência** de peritos que dominem o segmento do conhecimento que encerra o problema, pois é exatamente esse conhecimento que será o responsável direto pela resolução do problema;
- **existência** de tarefas que, para serem realizadas, necessitem da participação de vários especialistas que, isolados, não possuem conhecimentos suficientes



para realizá-la, ou seja, o conhecimento necessário para a análise e resolução do problema é multidisciplinar;

- **existência** de tarefas que requeiram conhecimento de detalhes que, se esquecidos, provocam a degradação do desempenho;
- **existência** de tarefas que demonstrem grandes diferenças entre o desempenho dos melhores e dos piores peritos;
- **escassez** de mão de obra especializada no conhecimento requerido para a solução do **problema**.

Com a emergência desta técnica, evidenciaram-se alguns importantes aspectos, até então inexplorados, por exemplo, o aumento significativo da produtividade de um especialista, na execução de tarefas especializadas, quando assistido por um sistema inteligente.

Outro aspecto relevante é a portabilidade destes sistemas especialistas, por serem passíveis de desenvolvimento e utilização em microcomputadores. Isto os torna bastante populares e acessíveis. Em geral, os sistemas com raciocínio automatizado podem ser utilizados incorporando bancos de dados já existentes na organização ou sendo incorporados ao conjunto de ferramentas disponíveis nos bancos de dados.

Sob nosso ponto de vista, a ciência da informação e muitas outras áreas podem encontrar, nos sistemas especialistas, eficientes ferramentas para o gerenciamento da informação. Disponibilizar ferramentas para suporte à tomada de decisão, neste caso, vai mais além do que fornecer gráficos e tabelas ao usuário: significa prestar-lhe orientação, na identificação de suas necessidades, simulando cenários e possibilitando maior exatidão e confiabilidade nos seus resultados.

Fonte: adaptado de Mendes (1997, on-line).



eu recomendo!



livro

Lógica e Design de Programação

Autora: Joyce Farrell

Editora: Cengage Learning

Ano: 2009

Sinopse: *Lógica e Design de Programação* é um guia de desenvolvimento de lógicas estruturadas para o programador iniciante. Sua leitura não pressupõe nenhuma experiência com linguagens de programação, nem conhecimento matemático avançado. A obra contém muitos exemplos provenientes da área de negócios, que ajudam os estudantes a obterem um conhecimento sólido em lógica, independentemente da linguagem de programação usada. A lógica usada no livro pode ser aplicada em qualquer linguagem de programação. Os fluxogramas, pseudocódigos e muitas ilustrações tornam o aprendizado mais fácil. O ícone “Não Faça Isso” destaca os erros mais comuns, ajudando os estudantes a evitá-los. A seção “Zona de Jogos” incentiva os leitores a criarem jogos usando os conceitos aprendidos. Além disso, há cinco apêndices, os quais permitem que os estudantes tenham experiências adicionais de estruturar grandes programas desestruturados, criar formulários de impressão, usar o sistema binário de numeração, trabalhar com grandes tabelas de decisões e testar softwares.





ESTRUTURAS DE REPETIÇÃO

PROFESSORES

Dra. Gislaine Camila Lapasini Leal
Me. Pietro Martins de Oliveira

PLANO DE ESTUDO

A seguir, apresentam-se as aulas que você estudará nesta unidade: • Estrutura de repetição • Estrutura for • Estrutura while • Estrutura do-while • Exercícios resolvidos.

OBJETIVOS DE APRENDIZAGEM

- Entender a estrutura de repetição • Compreender a estrutura for • Estudar a estrutura while • Compreender a estrutura do-while • Estudar os exercícios resolvidos.

INTRODUÇÃO



Nesta unidade, você estudará a construção de programas com a repetição de um trecho de código. A repetição permite executar um conjunto de instruções quantas vezes forem necessárias, sem precisar reescrever trechos de códigos idênticos.

É uma boa prática de programação evitar códigos repetitivos, incentivando o reuso, bem como a legibilidade e a manutenibilidade do código-fonte de um programa, independentemente da linguagem de programação. Sendo assim, a partir daqui, desenvolveremos um raciocínio voltado a identificar blocos de comandos que são repetitivos e cuja repetição pode ser automatizada por meio de estruturas de repetição.

Aprenderemos como utilizar as estruturas de repetição com laços contados e laços condicionais. O uso de laços contados é indicado em situações em que sabemos, previamente, quantas vezes as instruções precisam ser executadas. Na linguagem C, a estrutura de repetição com laço contado é o `for`, que possui, em sua estrutura, um campo para a inicialização de uma variável de controle, outro campo para a condição de parada da estrutura de repetição, e um terceiro campo, que permite incrementar ou decrementar a variável de controle.

Nos laços condicionais, não sabemos, previamente, o número de execuções e atrelamos a repetição a uma condição. Trataremos os casos com condição no início e no final do laço, estudando as estruturas: `while` e `do-while`.

Para facilitar o aprendizado, construiremos algoritmos utilizando cada uma dessas estruturas. Ao final desta unidade, você estará apto a construir programas com estruturas de repetição e poderá responder às seguintes questões: como repetir um trecho de código, em um número determinado de vezes? Como repetir um trecho de código com base em uma condição? Que estrutura de repetição é mais adequada para cada problema? Quando utilizar estruturas de repetição encadeadas?



1 ESTRUTURA DE REPETIÇÃO

Há situações, em nossos programas, em que precisamos repetir determinado trecho de código ou todo o código por várias vezes, em sequência. Nestes casos, utilizaremos uma estrutura de repetição que nos permite criar um *loop* para efetuar o processamento de um trecho de código quantas vezes forem necessárias. Na literatura, estas estruturas de repetição (*loops*) são, também, denominadas laços de repetição e malhas de repetição (MANZANO; OLIVEIRA, 1997).

A vantagem da estrutura de repetição é que não precisamos reescrever trechos de código idênticos, reduzindo, assim, o tamanho do algoritmo. Além disso, podemos determinar repetições com um número de vezes variável (LOPES; GARCIA, 2002).



conecte-se

Para saber um pouco mais sobre as estruturas de repetição em C, acesse:
http://pt.wikibooks.org/wiki/Programar_em_C/Estudo#LOOP_WHILE.

Fonte: os autores.



Nas estruturas de repetição, o número de repetições pode ser fixo ou estar relacionado a uma condição, isto é, os laços de repetição podem ser classificados em laços contados e condicionais. O laço contado é indicado para quando conhecemos, previamente, o número de iterações que precisa ser realizado. Em contrapartida,

em um laço condicional, este número de iterações é desconhecido e, para controlar o *loop*, utilizamos a estrutura condicional (ASCENCIO; CAMPOS, 2010).

Na linguagem C, utilizamos o comando **for** para laços contados e os comandos **while** e **do-while** para laços condicionais. Nas aulas seguintes, estudaremos cada uma destas estruturas de repetição da linguagem C, destacando suas sintaxe e aplicação.



pensando juntos

As estruturas **while** e **do-while** podem ser substituídas uma pela outra, além de poderem substituir, perfeitamente, a estrutura **for**. Você consegue imaginar alguma situação na qual esta afirmação não seja verdadeira?

AULA

ESTRUTURA FOR

```
for (<inicialização>;<condição>;<incremento>)
{
    <instruções>;
}
```

É uma estrutura do tipo laço contado, isto é, utilizada para um número definido de repetições. Intuitivamente, é mais comum utilizar esta estrutura quando sabemos, previamente, o número de vezes que o trecho de código precisa ser repetido. A sintaxe desta estrutura é:

Na primeira parte, temos a inicialização de uma variável, por exemplo, poderíamos ter algo como “`i=0 ;`”. A variável inicializada, aqui, tem, como função, controlar o número de repetições do laço. Essa inicialização é executada em primeiro lugar e uma única vez. A segunda parte consiste em uma expressão lógico-relacional que, ao assumir valor `falso`, determinará o fim da repetição. A terceira parte é responsável por atualizar (incrementar ou decrementar) o valor da variável utilizada para controlar a repetição. A atualização é executada ao fim de cada iteração (ASCENCIO; CAMPOS, 2010).

Quando temos a execução de apenas uma linha de comando, podemos suprimir as chaves. Neste caso, o compilador entenderá que a estrutura de repetição finaliza quando for encontrado o primeiro ponto e vírgula (`;`).

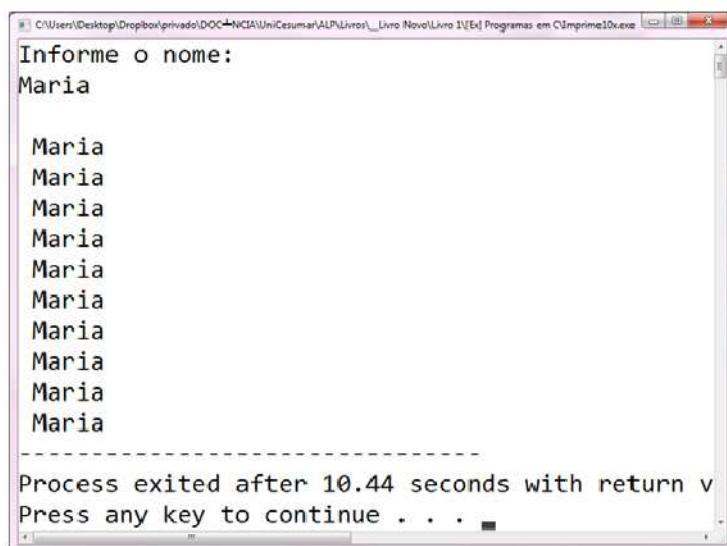
Para facilitar o entendimento, construiremos um programa que efetua a leitura de um nome e o imprime dez vezes na tela. Como sabemos, o número de iterações deve ser realizado utilizando a estrutura **for**, a expressão relacional que controla a execução do laço é `i<=10`, e temos a inicialização da variável `i` com o valor 1 e o incremento da variável `i` em uma unidade a cada execução do laço.

No Quadro 1, é apresentado o programa em C, que recebe um nome e o imprime dez vezes. Note que a linha da instrução **for** não possui ponto e vírgula ao final, e que a expressão relacional atua enquanto a expressão for verdadeira. Se utilizarmos, como condição, `i == 10`, o laço não será executado nenhuma vez. Faça este teste.

```
01 #include <stdio.h>
02 int main()
03 {
04     char nome[30];
05     int i;
06
07     printf("Informe o nome:\n");
08     scanf("%s", nome);
09     for(i=1; i<=10; i++)
10     {
11         printf("\n %s", nome);
12     }
13     return (0);
14 }
```

Quadro 1 - Programa em C / Fonte: os autores.

O resultado da simulação do programa (Quadro 1) é apresentado na Figura 1.



```
C:\Users\luis\Desktop\Dropbox\privado\DOC-NCIA\UnisCesumar\ALPU\livros\_\livro Novo\livro 1\Ex] Programas em C\primeiro10c.exe
Informe o nome:
Maria

Maria
Maria
Maria
Maria
Maria
Maria
Maria
Maria
Maria
-----
Process exited after 10.44 seconds with return value 0
Press any key to continue . . .
```

Figura 1 - Programa em C – Saída / Fonte: os autores.

Como temos a execução de uma única instrução, podemos suprimir as chaves, conforme programa apresentado no Quadro 2.

```
01 #include <stdio.h>
02 int main()
03 {
04     char nome[30];
05     int i;
06
07     printf("Informe o nome:\n");
08     scanf("%s", nome);
09     for(i=1; i<=10; i++)
10         printf("\n %s", nome);
11     return (0);
12 }
```

Quadro 2 - Programa em C / Fonte: os autores.

Lembre-se: só podemos suprimir as chaves nas situações em que queremos executar apenas uma instrução.



ESTRUTURA WHILE

A estrutura **while** é do tipo laço condicional, isto é, o *loop* baseia-se na análise de uma condição. Esta estrutura, geralmente, é utilizada quando temos um número indefinido de repetições e se caracteriza por realizar um teste condicional no início. Por causa deste último, podem ocorrer casos em que as instruções da estrutura de repetição nunca sejam executadas. Isso acontece quando o teste condicional da estrutura resulta em `falso` logo na primeira comparação (ASCENCIO; CAMPOS, 2010).

A sintaxe da estrutura **while** é:

```
while (<condição>)
{
    <instruções>;
}
```

Os comandos delimitados pelas chaves são executados enquanto a condição for verdadeira. Analisaremos o funcionamento desta estrutura utilizando o exemplo apresentado na aula anterior, receberemos um nome e o imprimiremos dez vezes.

Na estrutura **while**, temos que representar o critério de parada (dez iterações) utilizando uma condição. Para tanto, definimos uma variável que controlará o número de repetições. Esta variável precisa ser inicializada fora da estrutura de repetição e incrementada no interior do laço. No quadro a seguir, é apresentado o programa.

```

01 #include <stdio.h>
02 int main()
03 {
04     char nome[30];
05     int i;
06
07     printf("Informe o nome:\n");
08     scanf("%s", nome);
09     i = 0;
10     while (i != 10)
11     {
12         printf("\n %d - %s", i, nome);
13         i++;
14     }
15     return (0);
16 }
```

Quadro 3 - Programa em C / Fonte: os autores.

A figura a seguir ilustra o resultado da execução do programa, em que o laço é repetido até que a expressão relacional ($i \neq 10$) se torne falsa.

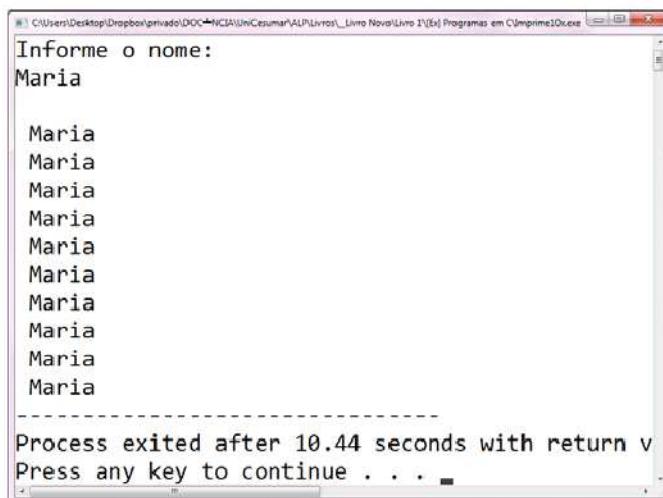


Figura 2 - Programa em C – Saída / Fonte: os autores.

Neste ponto, você pode estar se questionando por que a variável i não foi inicializada com o valor um em vez de zero. Você já sabe o porquê? Quando o valor de i é igual a dez, as instruções contidas no interior do laço não são executadas, pois a expressão relacional resulta em falso. Deste modo, teríamos a impressão do nome apenas nove vezes. Se quisermos inicializar a variável i em um, temos que alterar a expressão relacional para algo como $i \neq 11$.



ESTRUTURA DO-WHILE

A estrutura **do-while** é uma estrutura do tipo laço condicional, isto é, o *loop* baseia-se na análise de uma condição. Esta estrutura é utilizada quando temos um número indefinido de repetições e precisamos que o teste condicional seja realizado após a execução do trecho de código. Neste tipo de estrutura, o trecho de código é executado pelo menos uma vez, pois o teste condicional é realizado no fim (ASCENCIO; CAMPOS, 2010).

A sintaxe desta estrutura é dada por:

```
do
{
    <instruções>;
}
while (condição);
```

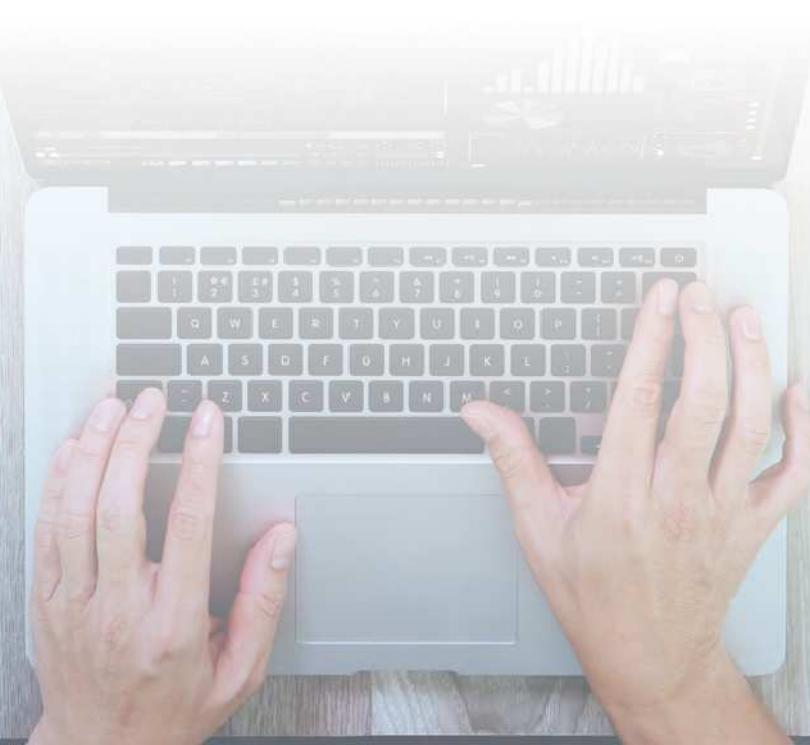
A diferença entre a estrutura **while** e **do-while** é o momento em que o teste condicional é analisado. No primeiro caso, temos a análise da condição e a execução do trecho de código apenas se o resultado do teste for verdadeiro. No segundo caso, temos a execução do trecho e, depois, a análise da condição, o que implica que o trecho de código será executado, no mínimo, uma vez.

Para esclarecer o funcionamento da estrutura, é apresentado, no quadro a seguir, o programa para o problema descrito na aula anterior, utilizando a estrutura **do-while**.

```
01 #include <stdio.h>
02 int main()
03 {
04     char nome[30];
05     int i;
06
07     printf("Informe o nome:\n");
08     scanf("%s", nome);
09     i = 0;
10     do
11     {
12         printf("\n %d - %s", i, nome);
13         i++;
14     }
15     while (i != 10);
16     return (0);
17 }
```

Quadro 4 - Programa em C / Fonte: os autores.

Se você ficou com dúvidas quanto ao funcionamento desta estrutura, fique tranquilo(a)! Veremos mais aplicações.



EXERCÍCIOS RESOLVIDOS

5



Problema 1

O problema consiste em ler um conjunto de números inteiros e contar a quantidade de números pares e ímpares. A leitura deve ser realizada até que seja lido o valor zero.

A entrada de dados consiste na leitura de números inteiros repetidas vezes, até que o valor zero seja digitado. O processamento é contar a quantidade de números pares e ímpares. E a saída é informar quantos dos números lidos na entrada são pares e quantos são ímpares. No Quadro 4, é apresentado um programa para este problema, utilizando a estrutura `while`, e no Quadro 5, utilizando a estrutura `do-while`.

```
01 #include <stdio.h>
02 int main()
03 {
04     int par, impar, num;
05     par = 0;
06     impar = 0;
07     printf("Informe o número:");
08     scanf("%d", &num);
09     while (num != 0)
10     {
11         if (num % 2 == 0)
```

```

12         par++;
13     else
14         impar++;
15     printf("Informe o número:\n");
16     scanf("%d", &num);
17 }
18 printf("A quantidade de par é : %d\n", par);
19 printf("A quantidade de ímpar é : %d\n", impar);
20 return (0);
21 }
```

Quadro 5 - Programa em C, Problema 1 / Fonte: os autores.

Note que, como a estrutura **while** realiza o teste no início, temos que efetuar a leitura do número antes do laço de repetição e, no interior do laço, também. O que aconteceria se não tivéssemos a leitura do número dentro do laço de repetição? Neste caso, teríamos um laço infinito, pois a condição num != 0 sempre resultaria em verdadeiro.

Na estrutura **do-while**, a leitura do número é realizada apenas no interior da estrutura de repetição, pois o teste é realizado ao final. Observe que, se o primeiro número fornecido pelo usuário for igual a zero, as instruções internas ao laço serão executadas, pois o teste é realizado, pelo menos, uma vez. Deste modo, teremos, como saída, a quantidade um para o número de pares.

```

01 #include <stdio.h>
02 int main()
03 {
04     int par, impar, num;
05     par = 0;
06     impar = 0;
07     do
08     {
09         printf("Informe o número:\n");
10         scanf("%d", &num);
11         if (num % 2 == 0)
12             par++;
13         else
14             impar++;
15     }
16     while (num != 0);
17     printf("A quantidade de par é: %d\n", par);
18     printf("A quantidade de ímpar é: %d\n", impar);
19     return (0);
20 }
```

Quadro 6 - Programa em C, Problema 1 (alternativo) / Fonte: os autores.

Agora que você já estudou como utilizar as estruturas de repetição baseadas em laço condicional, execute os dois programas e compare as saídas.

Fique atento(a)! No interior da estrutura de repetição **while** e **do-while**, precisamos ter uma instrução que altere o valor da expressão relacional. Caso contrário, o laço entrará em *loop*, sendo executado infinitamente.

Problema 2

O problema consiste em auxiliar um professor no fechamento das notas de uma turma. Para tanto, deve ser construído um programa que leia o código do aluno, o número de notas da disciplina e as notas. Calcule a média final de cada aluno e informe o número de alunos aprovados e reprovados. Para ser aprovado, o estudante precisa obter média maior ou igual a seis. O programa é encerrado quando é informado o código de aluno zero.

A entrada de dados consiste em ler o número de notas, o código do aluno e as notas. O processamento consiste em, a partir do número de notas informado para a disciplina, efetuar a repetição da leitura de notas, somá-las e calcular a média aritmética do aluno. Se a média for maior ou igual a seis, devemos incrementar a variável que controla o número de aprovados, senão temos que incrementar a variável que controla o número de reprovados. Como saída, temos o número de alunos aprovados e reprovados.

No Quadro 7, temos a resolução do problema utilizando a estrutura **do-while**, e no Quadro 8, com a estrutura **while**.

```
01 #include <stdio.h>
02 int main()
03 {
04     float nota, soma, media;
05     int cod, i, nnota, naprovado, nreprovado;
06
07     naprovado = 0;
08     nreprovado = 0;
09     printf("Informe o número de notas da
disciplina:\n");
10     scanf("%d", &nnota);
11     do
12     {
13         printf("Informe o código do aluno:\n");
```

```

14         scanf("%d", &cod);
15         soma = 0;
16         if (cod != 0)
17         {
18             for (i=1; i<=nnota; i++)
19             {
20                 printf("Informe a %d nota do
21                     aluno:\n", i);
22                 scanf("%f", &nota);
23                 soma = soma + nota;
24             }
25             media = soma/nnota;
26             if (media >=6)
27                 naprovado++;
28             else
29                 nreprovado++;
30         }
31         while (cod != 0);
32         printf("O número de aprovados é: %d\n",
33             naprovado);
34         printf("O número de reprovados é: %d\n",
35             nreprovado);
36         return (0);
37     }

```

Quadro 7 - Programa em C, Problema 2 / Fonte: os autores.

Observe que, ao utilizar a estrutura **do-while**, precisamos inserir uma condição que verifique se o código informado é diferente de zero, pois, como o teste condicional é executado ao final, teríamos a execução de todas as instruções de leitura de notas.

```

01 #include <stdio.h>
02 int main()
03 {
04     float nota, soma, media;
05     int cod, i, nnota, naprovado, nreprovado;
06     naprovado = 0;
07     nreprovado = 0;
08     printf("Informe o número de notas da
09         disciplina:\n");
10     scanf("%d", &nnota);
11     printf("Informe o código do aluno:\n");

```

```
11     scanf ("%d", &cod);
12     while (cod != 0)
13     {
14         soma = 0;
15         for (i=1; i<=nnota; i++)
16         {
17             printf("Informe a %d nota do aluno:\n",
18                   i);
19             scanf ("%f", &nota);
20             soma = soma + nota;
21         }
22         media = soma/nnota;
23         if (media >=6)
24             naprovado++;
25         else
26             nreprovado++;
27         printf ("Informe o código do aluno:\n");
28         scanf ("%d", &cod);
29     }
30     printf ("O número de aprovados é: %d\n",
31             naprovado);
32     printf ("O número de reprovados é: %d\n",
33             nreprovado);
34     return (0);
35 }
```

Quadro 8 - Programa em C, Problema 2 (alternativo) / Fonte: os autores.

Na construção utilizando a estrutura **while**, foi realizada a leitura do código do aluno antes da estrutura de repetição, pois a condição é testada no início do laço. Além disso, a leitura deve ser realizada no interior da estrutura para que o valor da expressão relacional possa ser alterado, senão teríamos um laço infinito.

Observe que utilizamos encadeamento de estruturas de repetição, isto é, uma estrutura de repetição dentro da outra. No primeiro caso, o encadeamento utilizou as estruturas **do-while** e **for** e, no segundo caso, **while** e **for**.

Ao utilizar encadeamento, tome cuidado para que todas as instruções da construção interna estejam embutidas na construção externa (MANZANO; OLIVEIRA, 1997; LOPES; GARCIA, 2002).

Problema 3

Faça um programa que leia um número inteiro e calcule o seu fatorial.

```

01 #include <stdio.h>
02 int main()
03 {
04     int num,i, fat;
05     printf ("Informe o número:\n");
06     scanf ("%d", &num);
07     fat = 1;
08     for (i=1; i <= num; i++)
09         fat = fat * i;
10     printf ("O fatorial é: %d\n", fat);
11     return (0);
12 }
```

Quadro 9 - Programa em C, Problema 3 / Fonte: os autores.

Problema 4

Elabore um programa que apresente todos os números divisíveis por três que sejam menores que 100.

```

01 #include <stdio.h>
02 int main()
03 {
04     int i;
05     for (i=3; i<=100; i=i+3)
06     {
07         printf ("%d \n", i);
08     }
09     return (0);
10 }
```

Quadro 10 - Programa em C, Problema 4 / Fonte: os autores.

Problema 5

Construa um programa que receba um número inteiro maior que um e verifique se ele é primo.

```
01 #include <stdio.h>
02 int main()
03 {
04     int num, i, qtdade;
05     printf("Informe o número:\n" );
06     scanf("%d", &num);
07     qtdade = 0;
08     for (i=1; i<=num; i++)
09     {
10         if (num % i == 0)
11             qtdade++;
12     }
13     if (qtdade == 2)
14         printf ("O número é primo.\n");
15     else
16         printf ("Não é primo.\n");
17 }
```

Quadro 11 - Programa em C, Problema 5 / Fonte: os autores.

Problema 6

A prefeitura de uma cidade está coletando informações sobre o salário e o número de filhos dos habitantes. A leitura de dados é realizada até que seja informado o valor -1 para o salário. Apresente a média de salário da população, a média de filhos e o maior salário.

```
01 #include <stdio.h>
02 int main()
03 {
04     int filhos, npessoas;
05     float salario, somas, somaf, msalario;
06     npessoas = 0;
07     somaf = 0;
08     somas = 0;
09     msalario = 0;
10     printf("Informe o salário:\n");
11     scanf("%f", &salario);
12     while (salario != -1)
13     {
14         printf("Informe o número de filhos:\n");
15         scanf("%d", &filhos);
16         npessoas++;
17         if (salario > msalario)
18             msalario = salario;
19         somaf = somaf + filhos;
20         somas = somas + salario;
21         printf("Informe o salário:\n");
22         scanf("%f", &salario);
23     }
24     printf("A média de salários e : %.2f\n",
25            somas/npessoas);
26     printf("A média de filhos e : %.2f\n",
27            somaf/npessoas);
28     printf("O maior salário e : %.2f\n", msalario);
29     return (0);
30 }
```

Quadro 12 - Programa em C, Problema 6 / Fonte: os autores.



Problema 7

Escreva um programa que receba a idade e a altura de várias pessoas, calcule e apresente a média de altura e idade das pessoas. A entrada de dados é encerrada quando for digitado o valor zero para a idade.

```
01 #include <stdio.h>
02 int main()
03 {
04     int idade, npessoas;
05     float altura, somaa, somai;
06     char sexo;
07     somaa = 0;
08     somai = 0;
09     npessoas = 0;
10     printf("Informe a idade:\n");
11     scanf("%d", &idade);
12     while (idade != 0)
13     {
14         printf("Informe a altura:\n");
15         scanf("%f", &altura);
16         npessoas++;
17         somai = somai + idade;
18         somaa = somaa + altura;
19         printf("Informe a idade:\n");
20         scanf("%d", &idade);
21     }
22     printf("A média de altura é: %.2f\n",
23            somaa/npessoas);
24     printf("A média de idade é: %.2f\n",
25            somai/npessoas);
26 }
```

Quadro 13 - Programa em C, Problema 7 / Fonte: os autores.

Problema 8

Em uma avaliação de um produto, o cliente responde sua opinião (1 – satisfatório; 2 – indiferente; 3 – insatisfatório). Faça um programa que leia idade e opinião e apresente: o número de clientes que responderam satisfatório, a média de idade dos clientes que opinaram como satisfatório, e o número de clientes que responderam insatisfatório. O programa se encerra quando for digitado o valor zero para idade.

```
01 #include <stdio.h>
02 int main()
03 {
04     int idade, npessoas, npessoasi;
05     int npessoass, opiniao;
06     float somai, media;
07     npessoas = 0;
08     npessoass = 0;
09     npessoasi = 0;
10     somai = 0;
11     printf("Informe a idade:\n");
12     scanf("%d", &idade);

13     while (idade != 0)
14     {
15         do
16         {
17             printf("Informe a opinião:\n");
18             scanf("%d", &opiniao);
19         }while ((opiniao!=1) && (opiniao!=2) &&
20 (opiniao!=3));

21         npessoas++;
22         if (opiniao == 1)
23         {
24             somai = somai + idade;
25             npessoass++;
26         }
27         else
28         {
29             if (opiniao == 3)
30                 npessoasi++;
31         }
32         printf("Informe a idade:\n");
33         scanf("%d", &idade);
34     }
35     media = (somai/npessoass);
36     printf ("O número de pessoas insatisfeitas é:
37     %d\n", npessoasi);
38     printf ("O número de pessoas satisfeitas é:
39     %d\n", npessoass);
40     printf ("A média de idade das pessoas satisfeitas
41 é: %.2f\n", media);
42     return (0);
43 }
```

Quadro 14 - Programa em C, Problema 8 / Fonte: os autores.

CONSIDERAÇÕES FINAIS

Nesta unidade, você aprendeu a construir algoritmos utilizando estruturas de repetição, que permitem a execução de um trecho de código repetidas vezes. Estas estruturas também são chamadas de laços de repetição.

Estudamos os laços de repetição contados e os laços condicionais. Nos laços de repetição contados, conhecemos a estrutura **for**, que é indicada nos casos em que sabemos quantas vezes o trecho de código precisa ser repetido. A sintaxe da estrutura for é algo como:

`for (<inicialização>; <condição>; <incremento>).` Nos laços de repetição condicionais, vimos as estruturas **while** e **do-while**. A estrutura **while** é utilizada quando não sabemos, previamente, o número de repetições que deve ser executado e impomos uma condição que é realizada no início. A sintaxe da estrutura while é algo como: `while (<condição>).`

A estrutura **do-while** é utilizada quando temos um número indefinido de repetições, no entanto, o teste lógico é realizado no final. Ela é similar ao comando while, todavia, independentemente do resultado lógico da condição de um do-while, o conjunto de instruções definido dentro dessa estrutura de repetição será executado ao menos uma vez. Estudamos que as estruturas baseadas em laços condicionais são mais flexíveis e que podem ser substituídas uma pela outra, isto é, podemos resolver um problema com algoritmo utilizando a estrutura **while** ou a estrutura **do-while**. Destaca-se que a estrutura **for** pode ser substituída pelo uso de estruturas baseadas em laços condicionais, todavia, pode ser mais intuitivo apelar para estruturas do tipo while em determinadas soluções algorítmicas.

Ao longo desta unidade, construímos algoritmos utilizando todos os conceitos aprendidos e, também, discutimos as particularidades de cada estrutura de repetição, enfatizando a forma de uso de cada uma delas.



1. Faça um programa que leia números inteiros até que seja informado o valor 0. Apresente a média dos valores, o maior e o menor valor e a quantidade de números pares e ímpares.
2. Para os seres humanos, trabalhar com séries numéricas sem o auxílio de ferramentas computacionais pode ser um grande transtorno. Assim sendo, construa um programa que leia o número de termos da série e imprima o valor de S .

$$S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \dots + \frac{1}{N}$$

3. Imagine que você esteja ensinando a tabuada para uma criança e precisa mostrar a ela todas as possíveis multiplicações entre os números que vão de um até dez. Assim, elabore um programa que imprima a tabuada do um ao dez.
4. É muito comum que programas tenham de implementar funcionalidades para identificar características específicas e realizar operações sobre um conjunto de dados. Dessa forma, faça um programa que apresenta a soma de todos os números inteiros ímpares entre 200 e 500.
5. Para poder classificar um conjunto de dados, é comum que programadores utilizem estruturas de decisão associadas a estruturas de repetição. Assim sendo, construa um programa que apresente todos os números divisíveis por três e por sete que sejam menores que 30.



6. Nada melhor do que uma máquina para realizar tarefas repetitivas, evitando esforço humano desnecessário. Assim sendo, elabore um programa que leia uma frase e o número de vezes que deseja imprimi-la.
7. Faça um programa que leia um conjunto de pedidos e calcule o total da compra. O pedido possui os seguintes campos: número, data (dia, mês e ano), preço unitário e quantidade. A entrada de pedidos é encerrada quando o usuário informa zero como número do pedido.
8. Elabore um programa que receba a idade, o peso, o sexo e o estado civil de várias pessoas e imprima a quantidade daquelas que são casadas, solteiras, separadas e viúvas. Apresente a média de idade e de peso. O algoritmo finaliza quando for informado o valor zero para idade.
9. Construa um programa que possibilite calcular a área total de uma residência (sala, cozinha, banheiro, quartos etc.). O programa deve solicitar a entrada do nome, a largura e o comprimento de determinado cômodo até que o nome do cômodo seja "FIM". O programa deve apresentar o valor total acumulado da área residencial.



ALGORITMOS RECURSIVOS

Você sabia que estruturas de repetição podem ser omitidas em alguns algoritmos repetitivos sem que, ainda assim, seja necessário replicar código-fonte desnecessariamente? É possível, de certa forma, “imitar” as iterações (repetições) dessas estruturas de repetição utilizando o conceito de recursividade. Para entender um pouco mais sobre a relação entre iterações e recursão, acompanhe o trecho do artigo que se segue.

“Um objeto é denominado recursivo quando sua definição é parcialmente feita em termos dele mesmo. A recursividade (ou recursão) é encontrada principalmente na matemática, mas está presente em algumas situações do cotidiano. Por exemplo, quando um objeto é colocado entre dois espelhos planos paralelos e frente a frente surge uma imagem recursiva, porque a imagem do objeto refletida num espelho passa a ser o objeto a ser refletido no outro espelho e, assim, sucessivamente.

Em programação, a recursividade é um mecanismo útil e poderoso que permite a uma função chamar a si mesma direta ou indiretamente, ou seja, uma função é dita recursiva se ela contém pelo menos uma chamada explícita ou implícita a si própria.

A ideia básica de um algoritmo recursivo consiste em diminuir sucessivamente o problema em um problema menor ou mais simples, até que o tamanho ou a simplicidade do problema reduzido permita resolvê-lo de forma direta, sem recorrer a si mesmo. Quando isso ocorre, diz-se que o algoritmo atingiu uma condição de parada, a qual



deve estar presente em pelo menos um local dentro algoritmo. Sem esta condição, o algoritmo não para de chamar a si mesmo, até estourar a capacidade da pilha, o que geralmente causa efeitos colaterais e até mesmo o término indesejável do programa.

Para todo algoritmo recursivo existe um outro correspondente iterativo (não recursivo), que executa a mesma tarefa. Implementar um algoritmo recursivo, partindo de uma definição recursiva do problema, em uma linguagem de programação de alto nível como Pascal e C é simples e quase imediato, pois o seu código é praticamente transscrito para a sintaxe da linguagem. Por essa razão, em geral, os algoritmos recursivos possuem código mais claro (legível) e mais compacto do que os correspondentes iterativos. Além disso, muitas vezes, é evidente a natureza recursiva do problema a ser resolvido, como é o caso de problemas envolvendo árvores – estruturas de dados naturalmente recursivas. Entretanto também há desvantagens:

- I - algoritmos recursivos quase sempre consomem mais recursos (especialmente memória, devido uso intensivo da pilha) do computador, logo tendem a apresentar um desempenho inferior aos iterativos;
- II - algoritmos recursivos são mais difíceis de serem depurados, especialmente quando for alta a profundidade de recursão (número máximo de chamadas simultâneas)".

Fonte: adaptado de Santos ([s. d.]).



eu recomendo!



livro

Programando em C

Autor: Ulysses de Oliveira

Editora: Ciência Moderna

Ano: 2008

Sinopse: descreve a linguagem C do modo como foi definida pelo padrão ANSI. Para auxiliar no aprendizado da programação em C, o livro faz uma introdução com exemplos para os programadores iniciantes na linguagem. Os exemplos são programas completos que, além de ensinarem a linguagem, ilustram algoritmos, estruturas de dados e técnicas de programação importantes. As principais rotinas de I/O são apresentadas e, também, inclui um utilíssimo Manual de Referência C.





VETORES, *STRINGS*, MATRIZES e *Structs*

PROFESSORES

Dra. Gislaine Camila Lapasini Leal
Me. Pietro Martins de Oliveira

PLANO DE ESTUDO

A seguir, apresentam-se as aulas que você estudará nesta unidade:

- Vetores e ordenação de vetores
- Pesquisa em vetores
- *Strings*
- Matrizes
- Estruturas (*struct*)
- Exercícios resolvidos

OBJETIVOS DE APRENDIZAGEM

- Estudar o conceito de vetores • Conhecer métodos de ordenação e pesquisa em vetores • Entender o conceito de *strings* e como manipulá-las • Estudar o conceito de matrizes e suas aplicações • Entender e aplicar estruturas (*structs*) • Observar os exercícios resolvidos.

INTRODUÇÃO



Nesta unidade, você estudará estruturas de dados homogêneas (vetores e matrizes), *strings* e estruturas de dados heterogêneas (*structs*).

As estruturas de dados homogêneas permitem a representação de diversas informações do mesmo tipo em uma única variável. Elas são divididas em estruturas de dados unidimensionais (por exemplo, os vetores) e estruturas multidimensionais (por exemplo, as matrizes). Os vetores são estruturas que podem ser controladas por um único índice, enquanto que as matrizes necessitam de dois índices ou mais para podermos realizar o acesso às suas posições e aos seus elementos.

Veremos como realizar a atribuição de valores às posições de uma estrutura homogênea, como acessar o conteúdo de dados armazenados nessas estruturas e, também, como preencher matrizes e vetores com dados informados pelo usuário. Em relação aos vetores, aprenderemos como realizar a sua ordenação bem como a pesquisa por determinado elemento, utilizando métodos tradicionais para isso.

Estudaremos as cadeias de caracteres, também conhecidas como *strings*, que são passagem de vetores do tipo `charactere`. Veremos como declarar e manipular este tipo de variável, que nos permite trabalhar com dados de texto. Veremos as estruturas de dados heterogêneas em linguagem C conhecidas como *structs*, que agregam informações de diversos tipos. Abordaremos, especificamente, como realizar atribuição, entrada, saída de dados e vetores de *structs*.

Ao término desta unidade, você saberá construir programas utilizando vetores, matrizes. Terá a capacidade de manipular *strings* contendo dados textuais e, também, poderá criar seus próprios tipos de dados a partir do uso de *structs*. Entenderá qual a importância destes conceitos e como utilizá-los em aplicações práticas.

Ao final da unidade, esperamos que você aprecie a experiência e expanda seu conhecimento. Vamos lá?



1 VETORES E ORDENAÇÃO de Vetores

Um vetor consiste em um arranjo de elementos armazenados, sequencialmente, na memória principal, todos acessíveis por meio de um mesmo nome (identificador). É um conjunto de variáveis de um mesmo tipo de dado, às quais são acessadas e referenciadas por meio de índices (LOPES; GARCIA, 2002).

Em C, os vetores são identificados pela existência de colchetes após o nome da variável no momento da declaração. O valor contido entre os colchetes representa o número de posições do vetor, ou seja, o seu tamanho (LOPES; GARCIA, 2002).

A declaração de um vetor é dada por:

```
<tipo_da_variável> <nome_da_variável>[<tamanho>];
```

Um vetor de inteiros pode ser declarado do seguinte modo, por exemplo:

```
int vetorA[20];
```

Neste caso, temos um vetor de inteiros denominado `vetorA`, o qual possui 20 posições, conforme ilustra a Figura 1. Este tipo de declaração faz com que seja reservado um espaço na memória suficientemente grande para armazenar o número de células especificadas em tamanho.

vetorA	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Figura 1 - Vetor / Fonte: os autores.

Fique atento(a)! Em C, a numeração das posições de um vetor inicia sempre em zero, ou seja, os índices utilizados para identificar as posições começam em zero e vão até o tamanho do vetor menos uma unidade. Podemos acessar cada componente de um vetor por meio de um índice associado ao identificador do vetor. Este índice pode ser referenciado diretamente ou por meio de uma expressão que resulte em um valor inteiro (ROCHA, 2006). Em nossos programas, devemos verificar se os valores atribuídos a um índice estão dentro dos limites definidos na declaração, pois o compilador não faz esta verificação, o que pode ocasionar em erros durante a execução do programa.

No Quadro 1, é apresentado um exemplo de manipulação de vetores em que é possível visualizar como é realizada a leitura e como são mostrados os elementos de um vetor.

```

01 #include <stdio.h>
02 int main()
03 {
04     int vetorA[10];
05     int i;
06
07     for (i=0;i<10;i++)
08     {
09         printf("Digite o %d elemento do vetor: \n",
10             i);
11         scanf("%d", &vetorA[i]);
12     }
13     printf ("Vetor preenchido\n");
14     for (i=0;i<10;i++)
15     {
16         printf("O elemento da posição %d é: %d \n",
17             i,
18             vetorA[i]);
19     }
20     return (0);
21 }
```

Quadro 1 - Programa em C / Fonte: os autores.

Neste exemplo, observe que o inteiro *i* é inicializado em zero (linhas 7 e 13) tanto para a operação de entrada de dados (linha 10) quanto para a exibição dos elementos do vetor (linha 15), pois a numeração de índices de um vetor sempre inicia em zero. Tanto a operação de leitura quanto a exibição é realizada por meio de laços de repetição.

O resultado da execução do programa é apresentado na Figura 2. O que aconteceria se lêssemos mais de dez elementos? O programa tentará ler normalmente, mas os escreverá em uma parte não alocada de memória, o que pode resultar em diversos erros durante a execução do programa.

```
C:\Users\Desktop\Dropbox\privado\DOC-NCIA\UniCesumarALP\livros\__Livro Novo\livro
Digite o 0 elemento do vetor:
10
Digite o 1 elemento do vetor:
20
Digite o 2 elemento do vetor:
30
Digite o 3 elemento do vetor:
40
Digite o 4 elemento do vetor:
50
Digite o 5 elemento do vetor:
60
Digite o 6 elemento do vetor:
70
Digite o 7 elemento do vetor:
80
Digite o 8 elemento do vetor:
90
Digite o 9 elemento do vetor:
100
Vetor preenchido
0 elemento da posição 0 é: 10
0 elemento da posição 1 é: 20
0 elemento da posição 2 é: 30
0 elemento da posição 3 é: 40
0 elemento da posição 4 é: 50
0 elemento da posição 5 é: 60
0 elemento da posição 6 é: 70
0 elemento da posição 7 é: 80
0 elemento da posição 8 é: 90
0 elemento da posição 9 é: 100
```

Figura 2 - Programa em C – Saída / Fonte: os autores.

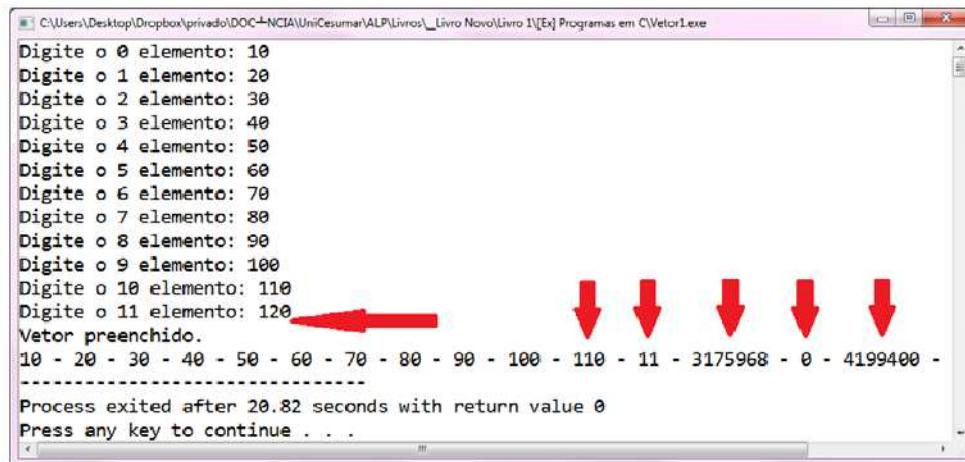
No quadro a seguir, temos um exemplo de manipulação de vetores em que a leitura e a exibição extrapolam o valor definido para o vetor. Repare que as linhas 6 e 12 permitem que seja realizado o acesso a posições do vetor superiores a dez, que é o tamanho máximo do vetor declarado na linha 4. Copie o código abaixo e compile. Note que a compilação é efetuada com sucesso, isto é, o compilador não verifica se estamos manipulando o vetor fora dos limites definidos na declaração.

```

01 #include <stdio.h>
02 int main()
03 {
04     int vetorA[10];
05     int i;
06     for (i=0;i<15;i++)
07     {
08         printf("\nDigite o %d elemento: ", i);
09         scanf("%d", &vetorA[i]);
10     }
11     printf ("Vetor preenchido.\n");
12     for (i=0;i<15;i++)
13     {
14         printf("%d - ", vetorA[i]);
15     }
16     return (0);
17 }
```

Quadro 2 - Programa em C / Fonte: os autores.

A figura a seguir ilustra o resultado da execução do programa (Quadro 2). Note que a leitura foi realizada normalmente. Na saída de dados, ocorreu um erro. Portanto, atente para os limites do vetor no momento de manipulá-lo.



```

C:\Users\Desktop\Dropbox\privado\DOC-NCIA\UniCesumar\ALP\Livros\_\Livro Novo\Livro 1\[Ex] Programas em C\Vetor1.exe
Digite o 0 elemento: 10
Digite o 1 elemento: 20
Digite o 2 elemento: 30
Digite o 3 elemento: 40
Digite o 4 elemento: 50
Digite o 5 elemento: 60
Digite o 6 elemento: 70
Digite o 7 elemento: 80
Digite o 8 elemento: 90
Digite o 9 elemento: 100
Digite o 10 elemento: 110
Digite o 11 elemento: 120
Vetor preenchido.
10 - 20 - 30 - 40 - 50 - 60 - 70 - 80 - 90 - 100 - 110 - 11 - 3175968 - 0 - 4199400 -
-----
Process exited after 20.82 seconds with return value 0
Press any key to continue . . .

```

Figura 3 - Programa em C – Saída /Fonte: os autores.

Repare que, de acordo as instruções das linhas de 6 a 10 do código-fonte do Quadro 2, o programa deveria tentar ler 15 elementos, mesmo tendo declarado um

vetor com dez posições. Como podemos observar na Figura 3, a execução desse mesmo trecho código-fonte se comportou de maneira não esperada, uma vez que permitiu a leitura do elemento das posições de índices iguais a 10 e 11 e, logo após, o laço de repetição da linha 6 é encerrado. Este comportamento adverso também pode ser observado na Figura 3, quando da exibição do conteúdo do vetor, uma vez que, nas posições que extrapolam o limite do vetor (índice i igual a 10, 11, 12, 13 e 14), temos um conteúdo esquisito, que, basicamente, se resume a lixo de memória.

Ordenação em vetor

A ordenação é o processo de rearranjar os elementos de acordo com um critério específico, com o objetivo de facilitar a localização (WIRTH, 1999). Na literatura, existem diversos métodos de ordenação, sendo o método da bolha (*Bubblesort*) o mais conhecido. O método da bolha consiste em percorrer o vetor repetidas vezes, comparando os elementos vizinhos. Se eles estão fora de ordem, é efetuada uma troca de posição.

No Quadro 3, temos um exemplo de ordenação utilizando o método da bolha. O primeiro laço de repetição (`for` da linha 6 à linha 10) efetua a leitura dos dez elementos do vetor. Em seguida, temos dois laços de repetição (`for`) aninhados (linhas 11 a 22), os quais são responsáveis por percorrer o vetor comparando os elementos vizinhos (posições `i` e `j`) e, se eles estão desordenados, é efetuada a troca de posição. Por fim, o terceiro laço (linhas 24 a 27) exibe os elementos do vetor já ordenados.

```
01 #include <stdio.h>
02 int main()
03 {
04     int vetorA[10];
05     int i, j, troca;
06     for (i=0;i<10;i++)
07     {
08         printf("Digite o %d elemento:", i);
09         scanf("%d", &vetorA[i]);
10     }
```

```
11     for (i=0; i<9; i++)
12     {
13         for (j=i+1; j<10;j++)
14         {
15             if (vetorA[i] > vetorA[j])
16             {
17                 troca = vetorA[i];
18                 vetorA[i] = vetorA[j];
19                 vetorA[j] = troca;
20             }
21         }
22     }
23     printf ("\n VETOR ORDENADO: \n");
24     for (i=0;i<10;i++)
25     {
26         printf("%d - ", vetorA[i]);
27     }
28     return (0);
29 }
```

Quadro 3 - Programa em C / Fonte: os autores.

A Figura 4 mostra a saída obtida na execução do programa (Quadro 3).

```
C:\Users\Desktop\Dropbox\privado\DOC-NCIA\UniCesumar\ALP\Livros\_Livro Novo\Livro...
Digite o 0 elemento:9
Digite o 1 elemento:10
Digite o 2 elemento:15
Digite o 3 elemento:27
Digite o 4 elemento:23
Digite o 5 elemento:1
Digite o 6 elemento:8
Digite o 7 elemento:0
Digite o 8 elemento:4
Digite o 9 elemento:3

VETOR ORDENADO:
0 - 1 - 3 - 4 - 8 - 9 - 10 - 15 - 23 - 27 -
-----
Process exited after 25.86 seconds with return value 0
Press any key to continue . . .
```

Figura 4 - Programa em C – Saída / Fonte: os autores.



2 PESQUISA EM VETOR

Em muitas situações, temos que pesquisar determinado elemento em um vetor. Quando temos um grande número de elementos, realizar a pesquisa manual é um processo inviável. Existem métodos que nos permitem verificar a existência de um valor dentro de um vetor (MANZANO; OLIVEIRA, 1997).

Estudaremos a pesquisa sequencial, que consiste em percorrer o vetor a partir do primeiro elemento até o último, sequencialmente, realizando testes lógicos e verificando se o elemento do vetor, posição a posição, é igual ao procurado. A pesquisa se encerra quando o elemento for encontrado ou quando todo o vetor foi percorrido por completo, sem que o elemento tenha sido encontrado (WIRTH, 1999).

```
01 #include <stdio.h>
02 int main()
03 {
04     int vetorA[10];
05     int i, acha, busca;
06     for (i=0;i<10;i++)
07     {
08         printf("Digite o %d elemento:", i);
09         scanf("%d", &vetorA[i]);
10     }
11     printf("Informe o elemento que deseja buscar:");
12     scanf("%d", &busca);
13     i = 0;
14     acha = 0;
15     while ((acha == 0) && (i < 10))
16     {
17         if (vetorA[i] == busca)
18         {
19             acha = 1;
20         }
21         else
22         {
23             i++;
24         }
25     }
26     if (acha == 1)
27         printf ("O elemento %d foi encontrado na posição
28             %d.", busca, i);
29     else
30         printf ("O elemento não foi encontrado.");
31 }
```

Quadro 4 - Programa em C / Fonte: os autores.

A implementação da pesquisa sequencial foi apresentada no Quadro 4. Agora que você conhece o funcionamento e a implementação da pesquisa sequencial, execute o programa a seguir e analise o resultado de sua saída.



STRINGS



A *string* consiste em uma cadeia de caracteres, um agregado de caracteres (vetor), terminando com o caractere especial '\0', que indica o fim da *string*. A declaração de uma *string* é dada por:

```
char <nome_string>[<tamanho>];
```

Note que não há um tipo *string* em C, mas sim, um vetor tipo *char*. Como ao final da *string* é armazenado o '\0', temos que declarar a *string* sempre com uma posição a mais do que o número de caracteres que desejamos armazenar. Além disso, devemos lembrar que as constantes *strings* aparecem entre aspas duplas e não é necessário acrescentar o '\0', isso é realizado em tempo de execução (ROCHA, 2006).

A linguagem C não possui um operador que atue com operandos do tipo *string*. Deste modo, a manipulação é realizada por meio de funções. No Quadro 5 são apresentadas algumas funções para a manipulação de *strings*.

Função	Operação
<code>strlen (x)</code>	Retorna o tamanho da <i>string</i> armazenada, sem '\0'.
<code>strcat (x, y)</code>	Concatena a <i>string</i> x à <i>string</i> y, sem alterar y.
<code>strcmp (x, y)</code>	Retorna a diferença (baseada na tabela ASCII) entre os dois primeiros caracteres diferentes, ou zero para igualdade entre as <i>strings</i> x e y.
<code>strcpy (x, y)</code>	Copia uma <i>string</i> em outra, ou seja, copia y em x.
<code>strncpy (x, y, n)</code>	Armazena em x os n primeiros caracteres de y.
<code>strlwr (x)</code>	Retorna o minúsculo da <i>string</i> x.
<code>strupr (x)</code>	Retorna o maiúsculo da <i>string</i> x.
<code>strstr (x, y)</code>	Verifica se y é subcadeia da <i>string</i> x.
<code>atoi (x)</code>	Converte x para o tipo int.
<code>atol (x)</code>	Converte x para o tipo long.
<code>atof (x)</code>	Converte x para o tipo float.
<code>gets (x)</code>	Lê caracteres até encontrar o de nova linha ('\n'), que é gerado quando se pressiona a tecla [enter].
<code>puts (x)</code>	Imprime uma <i>string</i> de cada vez.

Quadro 5 - Funções para a manipulação de *strings* / Fonte: os autores.

Fique tranquilo(a) quanto ao modo de utilizar essas funções, veremos exemplos utilizando as principais.

O Quadro 6 apresenta um programa que lê um nome completo utilizando a função **gets** e exibe o comprimento da *string*. O tamanho da *string* é obtido com a função **strlen**.

```
01 #include <stdio.h>
02 #include <string.h>
03 int main()
04 {
05     char nome[80];
06     int tamanho;
07     printf("Digite o seu nome completo:\n");
08     gets(nome);
09     tamanho = strlen(nome);
10     printf("O comprimento do nome é: %d\n", tamanho);
11     return (0);
12 }
```

Quadro 6 - Programa em C / Fonte: os autores.

A Figura 5 ilustra a saída obtida com a execução do programa (Quadro 5).

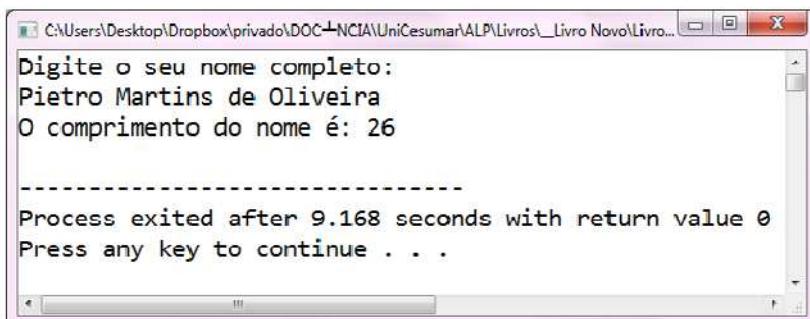


Figura 5 - Programa em C – Saída / Fonte: os autores.

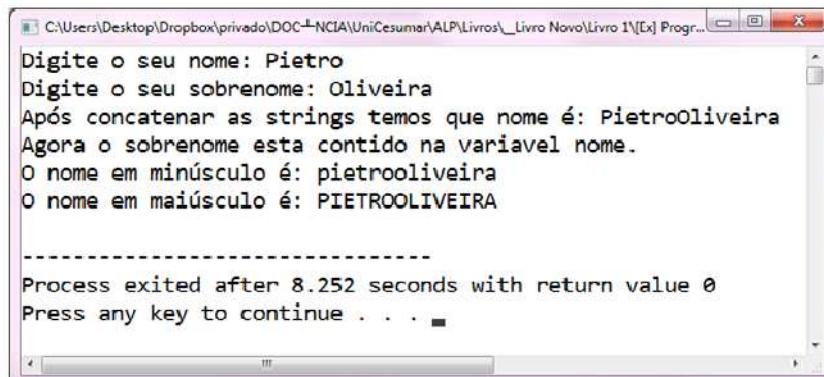
O programa do Quadro 7 demonstra o uso das funções **strcat**, **strstr**, **strupr** e **strlwr**.

```
01 #include <stdio.h>
02 #include <string.h>
03 int main()
04 {
05     char nome[80], sobrenome[80];
06     printf("Digite o seu nome: ");
07     gets(nome);
08     printf("Digite o seu sobrenome: ");
09     gets(sobrenome);
```

```
10     strcat(nome, sobrenome);
11     printf("Após concatenar as strings temos que nome
12         é: %s\n", nome);
13     if (strstr(sobrenome, nome) == 0)
14     {
15         printf("Agora o sobrenome está contido na
16             variável nome.\n");
17         strlwr(nome);
18         printf("O nome em minúsculo é: %s\n", nome);
19        strupr(nome);
20         printf("O nome em maiúsculo é: %s\n", nome);
21     }
22 }
```

Quadro 7 - Programa em C / Fonte: os autores.

O resultado da execução do programa (Quadro 7) é mostrado na Figura 6. Ao concatenar as duas *strings*, nome e sobrenome ficaram juntos. O que devemos fazer para que fique com um espaço entre o nome e o sobrenome? Antes de concatenar o valor da variável sobrenome, devemos concatenar um espaço em branco e, em seguida, o sobrenome.



```
C:\Users\Desktop\Dropbox\privado\DOC-NCIA\UniCesumar\ALP\Livros\__Livro Novo\Ex Programas> ./ex1
Digite o seu nome: Pietro
Digite o seu sobrenome: Oliveira
Após concatenar as strings temos que nome é: PietroOliveira
Agora o sobrenome esta contido na variavel nome.
O nome em minúsculo é: pietrooliveira
O nome em maiúsculo é: PIETROOLIVEIRA

-----
Process exited after 8.252 seconds with return value 0
Press any key to continue . . .
```

Figura 6 - Programa em C – Saída / Fonte: os autores.

Note que, quando utilizamos as funções para manipulação de *string*, devemos inserir o conteúdo da biblioteca <string.h>.



4 MATRIZES

Uma matriz é uma estrutura de dados tabular ou multidimensional, com o mesmo nome e alocada, sequencialmente, em memória. O acesso aos elementos da matriz é utilizando índices os quais podem ser referenciados diretamente ou por meio de uma expressão que resulte em um valor inteiro. Para cada dimensão da matriz, devemos ter um índice.

A sintaxe para a declaração de uma matriz é dada por:

```
<tipo> <nome> [<tamanho1>] [<tamanho2>] ... [<tamanhoN>];
```

Como exemplo de declaração de matrizes binárias, temos:

```
int matriz[2][3];
```

Neste caso, temos a declaração de uma matriz de inteiros composta por duas linhas e três colunas.

Nas operações de atribuição, leitura e escrita, devemos utilizar uma quantidade de laços de repetição aninhados compatível com a quantidade de dimensões da matriz. Isto é, uma matriz de duas dimensões deve ser controlada por dois laços de repetição aninhados, de três dimensões, três laços, e assim por diante (MANZANO; OLIVEIRA, 1997).

O programa do Quadro 8 exemplifica a leitura e a impressão de elementos de uma matriz.

```
01 #include <stdio.h>
02 int main()
03 {
04     int matrizA[2][10];
05     int i, j;
06     for (i=0;i<2;i++)
07     {
08         for (j=0;j<10;j++)
09         {
10             printf("Insira elemento da linha %d, coluna
11                 %d:", i, j);
12             scanf("%d", &matrizA[i][j]);
13         }
14         for (i=0;i<2;i++)
15         {
16             for (j=0;j<10;j++)
17             {
18                 printf("O elemento da linha %d, coluna %d é:
19                     %d\n", i, j, matrizA[i][j]);
20             }
21         }
22     }
}
```

Quadro 8 - Programa em C / Fonte: os autores.



5 ESTRUTURAS (*STRUCT*)

A utilização de vetores e matrizes implica em manipular uma grande quantidade de dados por meio de um acesso indexado, no entanto, até então, tínhamos a limitação de que todos os elementos deveriam ser do mesmo tipo. Em diversas situações, deparamo-nos com o fato de ter que armazenar informações relacionadas entre si, mas de tipos distintos (ROCHA, 2006). A estrutura permite agregar diversas informações, que podem ser de diferentes tipos. Possibilita gerar novos tipos de dados, além dos definidos pelas linguagens de programação (ASCENCIO; CAMPOS, 2010).

Em uma estrutura, o acesso aos elementos não é realizado por meio de sua localização, mas sim, por meio do nome do campo que se pretende acessar. Cada informação da estrutura é denominada de campo, o qual pode ser de diferentes tipos.

A sintaxe para a declaração de uma estrutura em C é:

```
struct <nome_da_estrutura>
{
    <tipo_de_dado1> <campo1>;
    <tipo_de_dado2> <campo2>;
    ...
    <tipo_de_dadoN> <campoN>;
};
```

A partir da definição da estrutura, o programa pode considerar que existe um novo tipo de dado a ser utilizado, reconhecido pelo <nome_da_estrutura>. Este novo tipo de dado é capaz de armazenar informações que podem ser de tipos distintos, de acordo com a forma que os <tipo_de_dados> foram estabelecidos para cada <campo> (ASCENCIO; CAMPOS, 2010).

Consideremos, como exemplo, a definição de uma estrutura que armazena a ficha de um produto, conforme ilustra a Figura 7. As informações a serem armazenadas são o código do produto, a descrição, o preço e o saldo em estoque, as quais representam os campos que a estrutura deve conter.

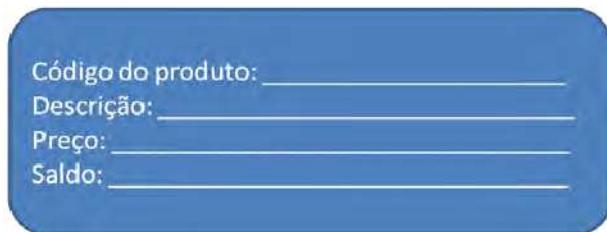


Figura 7 - Ficha de produto / Fonte: os autores.

Um ícone de um cérebro branco no lado esquerdo de uma barra azul. Sobre a barra, a frase "pensando juntos" está escrita em um fonte cinza.

Funções como `flush(stdin)`, `system("cls")` e `system("pause")` são muito utilizadas na programação em linguagem C. Você saberia informar para que servem tais funções?

A declaração da estrutura para a ficha do produto seria algo como:

```
struct produto
{
    int codigo;
    char descricao[50];
    float preco;
    int saldo;
};
```

Esta declaração indica que o programa poderá utilizar um novo tipo de dado identificado como `struct produto`, que contém quatro informações (campos). Ascencio e Campos (2010) destacam que a `struct` só pode ser utilizada

dentro do bloco em que foi definida. Isto é, uma `struct` definida dentro das chaves que delimitam a função `main` só poderá ser utilizada por variáveis que também estejam neste bloco. Para que a `struct` seja acessível de qualquer parte do programa, temos que defini-la fora da função `main` ou de qualquer outra função, preferencialmente, após os comandos `#include`.

Para utilizar uma `struct`, temos que declarar uma variável deste tipo do seguinte modo:

```
struct <nome_da_estrutura> <nome_da_variável>;
```

No caso da ficha de produto, teríamos a seguinte declaração:

```
struct produto ficha;
```

Esta declaração nos indica que temos uma variável denominada `ficha`, a qual é do tipo `struct produto`. Nas operações de atribuição, leitura e escrita, utilizamos o nome da variável `struct` e seu campo correspondente separado por um caractere “.” (ponto).

No Quadro 9, temos um programa que exemplifica as operações de leitura e escrita utilizando `struct`. Observe que, inicialmente, temos a declaração da `struct` após a diretiva `include` e, no interior da função `main`, temos, na linha 12, a declaração de uma variável do tipo `struct produto`. O acesso a cada um dos campos é realizado por meio do nome da variável mais o caractere ponto e o nome do campo, como podemos ver nas linhas 15, 17, 19, 21, 23, 24, 25 e 26.

```
01 #include <stdio.h>
02 struct produto
03 {
04     int    codigo;
05     char   descricao[50];
06     float  preco;
07     int    saldo;
08 };
09
10 int main()
11 {
12     struct produto ficha;
13
14     printf("Digite o código do produto:");
```

```
15     scanf("%d", &ficha.codigo);
16     printf("Digite a descrição do produto:");
17     scanf("%s", ficha.descricao);
18     printf("Digite o preço do produto:");
19     scanf("%f", &ficha.preco);
20     printf("Digite o saldo do produto:");
21     scanf("%d", &ficha.saldo);
22
23     printf("Código : %d\n", ficha.codigo);
24     printf("Descrição : %s\n", ficha.descricao);
25     printf("Preço : %.2f\n", ficha.preco);
26     printf("Saldo : %d\n", ficha.saldo);
27
28     return (0);
29 }
```

Quadro 9 - Programa em C / Fonte: os autores.

Este exemplo ilustra a leitura e a escrita da ficha de um produto. E, se quisermos armazenar a ficha de dez produtos, precisamos criar dez variáveis do tipo struct produto? Não, podemos criar um vetor de struct.

```
01 #include <stdio.h>
02
03 struct produto
04 {
05     int    codigo;
06     char   descricao[50];
07     float  preco;
08     int    saldo;
09 };
10
11 int main()
12 {
13     struct produto ficha[10];
14     int i;
15
16     for (i=0; i<10; i++)
17     {
```

```
18     printf("Digite o código do produto:");
19     scanf("%d", &ficha[i].codigo);
20     printf("Digite a descrição do produto:");
21     scanf("%s", ficha[i].descricao);
22     printf("Digite o preço do produto:");
23     scanf("%f", &ficha[i].preco);
24     printf("Digite o saldo do produto:");
25     scanf("%d", &ficha[i].saldo);
26 }
27
28 for (i=0; i<10;i++)
29 {
30     printf ("\n PRODUTO %d\n", i+1);
31     printf(" Código : %d\n", ficha[i].codigo);
32     printf(" Descrição : %s\n",
33            ficha[i].descricao);
34     printf(" Preço : %.2f\n", ficha[i].preco);
35     printf(" Saldo : %d\n", ficha[i].saldo);
36 }
37 }
```

Quadro 10 - Programa em C / Fonte: os autores.

O Quadro 10 exemplificou o uso de um vetor de **struct** com operações de leitura e escrita.



conecte-se

Para saber um pouco mais sobre ordenação pelo método da bolha, acesse os vídeos disponíveis em:

<http://www.youtube.com/watch?v=PDeTKL68jyE&feature=related>

<http://www.youtube.com/watch?v=rCKbfdvnhl&feature=related>

Para saber um pouco mais sobre declaração e construção de *structs*, acesse:

http://pt.wikibooks.org/wiki/Programar_em_C/Estruturas

Fonte: os autores.



EXERCÍCIOS RESOLVIDOS

Problema 1

O problema consiste em elaborar um cadastro para 20 livros, contendo as seguintes informações: código, título, autor, área, ano e editora. Desenvolver um menu com as seguintes opções:

1. Cadastrar os livros.
2. Imprimir as informações dos livros.
3. Pesquisar livros por código.
4. Ordenar os livros por ano.
5. Sair do programa.

No Quadro 11, temos o programa para o problema descrito. Na solução, empregamos o conceito de `struct` para criar a ficha do livro, vetor de `struct` para armazenar as informações dos 20 livros, o método de pesquisa sequencial para efetuar a busca de um livro por código e o método de ordenação da bolha para classificar os livros de acordo com o ano.

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 #define TAM 20
05
06 struct livro
07 {
08     int    codigo;
09     char   titulo[50];
10     char   autor[30];
11     char   area[30];
12     int    ano;
13     char   editora[30];
14 };
15
16 int main()
17 {
18     struct livro ficha[TAM];
19     struct livro troca;
20     int busca, i, j, acha, op;
21
22     op = 0;
23     while  (op !=5)
24     {
25         printf("1 - Cadastrar os livros\n");
26         printf("2 - Imprimir os livros
27 cadastrados\n");
28         printf("3 - Pesquisar livros por código\n");
29         printf("4 - Ordenar os livros por ano\n");
30         printf("5 - Sair\n");
31         printf("Digite a opção desejada: ");
32         scanf("%d", &op);
33         fflush(stdin);
34         if (op == 1)
35         {
36             system("cls");
37             for (i=0; i<TAM; i++)
38             {
39                 printf("Digite o código do livro da
40                 posição %d:", i+1 );
41                 scanf("%d", &ficha[i].codigo);
42                 fflush(stdin);
43                 printf("Digite o título do livro: ");
44                 scanf("%50[^\\n]s", &ficha[i].titulo);
45                 fflush(stdin);
46                 printf("Digite o nome do autor: ");
47                 scanf("%30[^\\n]s", &ficha[i].autor);
```

```
46         fflush(stdin);
47         printf("Digite a área do livro: ");
48         scanf("%30[^\\n]s", &ficha[i].area);
49         fflush(stdin);
50         printf("Digite o ano: ");
51         scanf("%d", &ficha[i].ano);
52         fflush(stdin);
53         printf("Digite o nome da editora: ");
54         scanf("%30[^\\n]s", &ficha[i].editora);
55         fflush(stdin);
56     }
57 }
58 else
59 {
60     if (op == 2)
61     {
62         system("cls");
63         for (i=0; i<TAM; i++)
64         {
65             printf("\nCÓDIGO: %d\n",
66                 ficha[i].codigo );
67             printf("TÍTULO: %s\n",
68                 ficha[i].titulo);
69             printf("AUTOR: %s\n",
70                 ficha[i].autor);
71             printf("ÁREA: %s\n", ficha[i].area);
72             printf("ANO: %d\n", ficha[i].ano);
73             printf("EDITORA: %s\n\n",
74                 ficha[i].editora);
75         }
76     }
77     else
78     {
79         if (op == 3)
80         {
81             system("cls");
82             printf("Digite o código que deseja
83             buscar: ");
84             scanf ("%d", &busca);
85             i = 0;
86             acha = 0;
87             while ((i < TAM) && (acha == 0))
88             {
89                 if (ficha[i].codigo == busca)
90                     acha = 1;
91                 else
```

```
87             i++;
88         }
89         if (acha == 1)
90         {
91             printf("\nCÓDIGO: %d\n",
92                 ficha[i].codigo );
93             printf("TÍTULO: %s\n",
94                 ficha[i].titulo);
95             printf("AUTOR: %s\n",
96                 ficha[i].autor);
97             printf("ÁREA: %s\n",
98                 ficha[i].area);
99             printf("ANO: %d\n", ficha[i].ano);
100            printf("EDITORIA: %s\n\n",
101                ficha[i].editora);
102        }
103    else
104    {
105        printf("\n Registro não
106        encontrado");
107    }
108}
109else
110{
111    if (op ==4)
112    {
113        system("cls");
114        for (i=0;i<TAM-1;i++)
115        {
116            for (j=i+1;j<TAM;j++)
117            {
118                if (ficha[i].ano >
119                    ficha[j].ano)
120                {
```

```
121 }  
122 }  
123 }  
124 }  
125 }  
126 }  
127 return (0);  
128 }
```

Quadro 11 - Programa em C, Problema 1 / Fonte: os autores.

Observe que utilizamos a função `system()` da biblioteca `stdlib.h`, incluída ao programa na linha 2. Esta função é responsável por realizar chamadas de sistema para executar o que for passado por parâmetro. No caso da invocação da função `system("cls")` nas linhas 35, 62, 77 e 105, queremos invocar o sistema operacional (nesse caso, o MS-DOS - Windows) para que ele execute o comando “`cls`” no terminal em que o programa é executado, com o intuito de limpar a tela, como se fosse um *clear screen*. Para que o mesmo efeito seja alcançado em ambientes Linux (UNIX), devemos invocar o comando da seguinte forma: `system("clear");`.

Além disso, temos a invocação da função `fflush(stdin)` sendo invocada após toda e qualquer operação de entrada de dados (`scanf` das linhas 31, 39, 42, 45, 48, 51 e 54). Esta função é responsável por “limpar” o *buffer* do teclado (indicado pelo parâmetro `stdin`). Fazemos isso pois, em alguns dispositivos computacionais, é possível que operações de leitura de dados deem problema em função do *buffer* do teclado armazenar lixos de memória.

Repare, ainda, que os `scanf` das linhas 42, 45, 48 e 54 contêm um especificador de formato similar a “`%50[^\\n]s`”. Nesse caso, o especificador de formato, da maneira como está posta (“`%50[^\\n]s`”), permite que o `scanf` leia apenas, no máximo, 50 caracteres, ou até que a tecla ‘`\n`’ seja pressionada, além de permitir que sejam lidos espaços.

É importante que você execute este programa e analise o seu funcionamento com cautela, pois, na construção do programa, utilizamos todos os conceitos vistos no decorrer desta unidade, somados a comandos novos, que são muito úteis.

Problema 2

Escreva um programa que leia um vetor com 30 elementos inteiros e escreva-os em ordem contrária à da leitura.

```
01 #include <stdio.h>
02 int main()
03 {
04     int vetorA[30];
05     int i;
06
07     for (i=0;i<30;i++)
08     {
09         printf("Digite o %d elemento:", i);
10         scanf("%d", &vetorA[i]);
11     }
12     for (i=29;i>=0;i--)
13     {
14         printf("\n %d", vetorA[i]);
15     }
16     return (0);
17 }
```

Quadro 12 - Programa em C, Problema 2 / Fonte: os autores.

Problema 3

Faça um programa que leia dois vetores A e B, com 20 números inteiros. Efetue a soma dos dois vetores em um vetor C e imprima o vetor C em ordem crescente.

```
01 #include <stdio.h>
02 int main()
03 {
04     int vetorA[30], vetorB[30], vetorC[30];
05     int i, j, troca;
06     for (i=0;i<30;i++)
07     {
08         printf("Digite o %d elemento do vetor A: ", i);
09         scanf("%d", &vetorA[i]);
10         printf("Digite o %d elemento do vetor B: ", i);
11         scanf("%d", &vetorB[i]);
```

```

12         vetorC[i] = vetorA[i] + vetorB[i];
13     }
14     for (i=0;i<29;i++)
15     {
16         for (j=i+1; j<30; j++)
17         {
18             if (vetorC[i] > vetorC[j])
19             {
20                 troca = vetorC[i];
21                 vetorC[i] = vetorC[j];
22                 vetorC[j] = troca;
23             }
24         }
25     }
26     for (i=0;i<30;i++)
27     {
28         printf("%d - ", vetorC[i]);
29     }
30     return (0);
31 }
```

Quadro 13 - Programa em C, Problema 3 / Fonte: os autores.

Problema 4

Faça um programa que leia um nome e apresente as letras que se encontram nas posições pares.

```

01 #include <stdio.h>
02 #include <string.h>
03 int main()
04 {
05     char nome[30];
06     int tam, i;
07     printf("Digite o nome: ");
08     gets(nome);
09     tam = strlen(nome);
10     for (i=0;i<tam; i++)
11     {
12         if (i % 2 == 0)
13             printf ("\n %c", nome[i]);
14     }
15     return (0);
16 }
```

Quadro 14 - Programa em C, Problema 4 / Fonte: os autores.

Problema 5

Construa um programa que leia uma palavra e a escreva de trás para frente.

```
01 #include <stdio.h>
02 #include <string.h>
03 int main()
04 {
05     char palavra[30];
06     int tam, i;
07     printf ("Digite a palavra:");
08     gets(palavra);
09     tam = strlen(palavra);
10     for (i=tam-1;i>=0; i--)
11     {
12         printf ("%c", palavra[i]);
13     }
14     return (0);
15 }
```

Quadro 15 - Programa em C, Problema 5 / Fonte: os autores.

Problema 6

Faça um programa que leia uma palavra e a imprima quantas vezes forem o número de caracteres.

```
01 #include <stdio.h>
02 #include <string.h>
03 int main()
04 {
05     char palavra[30];
06     int tam, i;
07     printf ("Digite o nome: ");
08     gets(palavra);
09     tam = strlen(palavra);
10     for (i=0;i<tam; i++)
11     {
12         printf ("%s\n", palavra);
13     }
14     return (0);
15 }
```

Quadro 16 - Programa em C, Problema 6 / Fonte: os autores.

Problema 7

Construa um programa que efetue a leitura de quatro notas de 20 alunos, calcule a média de cada aluno e a média da turma.

```
01 #include <stdio.h>
02 #define TAM 20
03 int main()
04 {
05     float media[TAM];
06     float notas[TAM][4];
07     float somat, mediat, soma;
08     int i, j;
09
10     somat = 0;
11     mediat = 0;
12     for (i=0; i<TAM; i++)
13     {
14         soma = 0;
15         for (j=0; j<4; j++)
16         {
17             printf("Informe a nota %d do aluno %d: ",
18                   j+1, i+1);
19             scanf("%f", &notas[i][j]);
20             soma = soma + notas[i][j];
21         }
22         media[i] = soma/4;
23         somat = somat + media[i];
24     }
25     mediat = somat/TAM;
26     for (i=0; i<TAM; i++)
27     {
28         printf("A média do aluno %d é: %.2f\n", i,
29               media[i]);
30     }
31     printf("A média da turma é: %.2f\n", mediat);
32     return (0);
33 }
```

Quadro 17 - Programa em C, Problema 7 / Fonte: os autores.

Problema 8

Construa um programa que leia informações (matrícula, nome, setor e salário) de 20 funcionários. Deve ser permitido executar quantas consultas o operador desejar, em que ele digita a matrícula e são apresentados o setor e o salário. Se a matrícula digitada não existir, informar o usuário.

```
01 #include <stdio.h>
02 #define TAM 20
03 struct funcionario
04 {
05     int    matricula;
06     char   nome[50];
07     char   setor[30];
08     float  salario;
09 };
10
11 int main()
12 {
13     struct funcionario ficha[TAM];
14     int busca, i, acha;
15     char op;
16
17     for (i=0; i<TAM; i++)
18     {
19         printf("Digite a matrícula do funcionário: ");
20         scanf("%d", &ficha[i].matricula);
21         fflush(stdin);
22         printf("Digite o nome: ");
23         scanf("%50[^\\n]s", &ficha[i].nome);
24         fflush(stdin);
25         printf("Digite o setor: ");
26         scanf("%30[^\\n]s", &ficha[i].setor);
27         fflush(stdin);
28         printf("Digite o salário: ");
29         scanf("%f", &ficha[i].salario);
30         fflush(stdin);
31     }
32
33     do{
34         printf("Deseja realizar busca (S/N): ");
```

```
35         scanf("%c", &op);
36         fflush(stdin);
37     }while ((op != 'S') && (op !='s') && (op !='n') &&
38           (op !='N'));
39
40     while ((op=='S') || (op=='s'))
41     {
42         printf("Informe a matrícula que deseja buscar:
43             ");
44         scanf("%d", &busca);
45         fflush(stdin);
46         i = 0;
47         acha = 0;
48         while ((i<TAM) && (acha==0))
49         {
50             if (ficha[i].matricula == busca)
51                 acha = 1;
52             else
53                 i++;
54         }
55         if (acha ==1)
56         {
57             printf("O setor é: %s\n", ficha[i].setor);
58             printf("O salário é: %.2f\n",
59                   ficha[i].salario);
60         }
61         else
62         {
63             printf("Matrícula não cadastrada\n");
64         }
65
66         do{
67             printf("Deseja realizar busca (S/N):");
68             scanf("%c", &op);
69             fflush(stdin);
70         }while ((op != 'S') && (op !='s') && (op !='n') &&
71           (op !='N'));
72
73     }
74
75     return (0);
76 }
```

Quadro 18 - Programa em C, Problema 8 / Fonte: os autores.

CONSIDERAÇÕES FINAIS

Nesta unidade, você aprendeu a construir programas utilizando vetores, *strings*, matrizes e estruturas. Vimos que vetores e matrizes são estruturas de dados homogêneas que agrupam diversas informações, do mesmo tipo, em uma única variável. O acesso a essas estruturas é indexado, de modo que, nos vetores, temos um único índice, e no caso de matrizes, o número de dimensões determina o número de índices.

Aprendemos como efetuar classificação e pesquisa em vetores. O método de classificação estudado foi o *Bubblesort*, que varre o vetor repetidas vezes, comparando os elementos vizinhos. Se eles estão fora de ordem, é efetuada uma troca de posição. Este método é utilizado quando queremos rearranjar o vetor segundo algum critério. No que se refere à pesquisa, vimos o método de pesquisa sequencial, que nos permite verificar se um dado elemento encontra-se no vetor ou não.

Estudamos o conceito de *strings*, como efetuar leitura e escrita e, também, conhecemos funções que permitem concatenar, comparar, copiar, armazenar, imprimir, converter para um valor numérico, dentre outros. Em relação às *strings*, não podemos esquecer de declará-la com uma posição a mais que o número de caracteres que desejamos armazenar, pois, ao final, é inserido o "\0" que indica o fim da *string*.

Trabalhamos a definição de novos tipos de dados utilizando estruturas (*structs*), que são capazes de armazenar informações de tipos diferentes. Estudamos como manipular estas estruturas e como construir vetores delas.

No decorrer desta unidade, revisamos os conceitos vistos nas unidades anteriores, principalmente, as estruturas de repetição, pois elas são utilizadas nas operações de atribuição, leitura e escrita. Por fim, construímos programas envolvendo todos os conceitos abordados. A unidade está acabando, mas nos vemos na próxima. Até lá!



1. A computação, frequentemente, é utilizada para servir de ferramenta na identificação de diferenças ou semelhanças entre objetos. Dessa forma, faça um programa que leia dois vetores **A** e **B** e apresente a quantidade de posições que possuem elementos diferentes entre um vetor e outro.
2. É comum que um programa seja capaz de realizar a ordenação de elementos ou dados de forma a facilitar as operações futuras e a apresentação deste conjunto de dados. Assim sendo, escreva um programa que leia um vetor A e o apresente em ordem decrescente.
3. O processamento de dados textuais, atualmente, é bastante avançado, e já existem bibliotecas capazes de auxiliar neste tipo de situação. Elabore um programa que leia uma palavra e, se ela tiver número ímpar de caracteres, imprima todas as suas vogais.
4. Um dos principais benefícios de aprender a programar é o fato de que é possível automatizar tarefas repetitivas. Dessa forma, faça um programa que leia uma palavra e o número de vezes que se deseja imprimi-la.
5. Um programador que se preze necessita dominar vários tipos de estruturas de dados. Uma das estruturas mais importantes, em programas, são as matrizes. Assim, construa um programa que recebe duas matrizes inteiras de ordem 5 e imprima a soma e a diferença entre elas.



6. Já pensou em facilitar a vida do seu professor, criando um programa que automatize os cálculos das notas de seus **alunos**? Sendo assim, faça um programa que efetue a leitura dos nomes de cinco alunos e, também, de suas quatro notas bimestrais. Calcule a média de cada aluno e apresente os nomes classificados em ordem crescente de média.
7. É possível criar sistemas para organização e controle de estoque nas mais diversas áreas. Assim sendo, elabore um programa para efetuar o cadastro de 20 livros e imprimi-los. O cadastro deve conter as seguintes informações: título, autor, editora, edição e ano.
8. Com o surgimento dos smartphones, a funcionalidade de agenda eletrônica passou a fazer parte do cotidiano das pessoas, de modo geral. Pensando em um protótipo de agenda, faça um programa para efetuar o cadastro de 30 contatos. O cadastro deve conter as seguintes informações: nome, telefone e e-mail. Apresente todos os cadastros.



Para além de simples vetores, matrizes e registros, é possível desenvolver estruturas de dados altamente complexas a partir dos conceitos vistos nesta unidade. Um exemplo de estrutura de dados não tão complexa, mas que pode ser implementada por meio de vetores, são as pilhas. O artigo que se segue mostra uma maneira simples de implementar pilhas em linguagem C.

Uma pilha é uma estrutura de dados que admite remoção de elementos e inserção de novos objetos. Mais especificamente, uma pilha (*stack*) é uma estrutura sujeita à seguinte regra de operação: sempre que houver uma remoção, o elemento removido é o que está na estrutura há menos tempo.

Em outras palavras, o primeiro objeto a ser inserido na pilha é o último a ser removido. Esta política é conhecida pela sigla LIFO (*Last-In-First-Out*).

IMPLEMENTAÇÃO EM UM VETOR

Suponha que nossa pilha está armazenada em um vetor `pilha[0..N-1]`. (A natureza dos elementos do vetor é irrelevante: eles podem ser inteiros, bytes, ponteiros, etc.) Digamos que a parte do vetor ocupada pela pilha é

pilha[0..t-1] .

O índice t indica a primeira posição vaga da pilha e $t-1$ é o índice do topo da pilha. A pilha está vazia se t vale 0 e cheia se t vale N . No exemplo da figura, os caracteres A, B, ..., H foram inseridos na pilha nessa ordem:



Para *remover*, ou *tirar*, um elemento da pilha — essa operação é conhecida como *desempilhar* (*to pop*) — faça



```
x = pilha[--t];
```

Isso equivale ao par de instruções `t -= 1; x = pilha[t];`, nessa ordem. É claro que você só deve desempilhar se tiver certeza de que a pilha não está vazia.

Para *inserir*, ou *colocar*, um objeto `y` na pilha — a operação é conhecida como empilhar (*to push*) — faça

```
pilha[t++] = y;
```

Isso equivale ao par de instruções `pilha[t] = y; t += 1;`, nessa ordem. Antes de empilhar, certifique-se de que a pilha não está cheia, para evitar um *transbordamento (overflow)*.

Para facilitar a leitura do código, é conveniente embalar essas operações em duas pequenas funções. Se os objetos com que estamos lidando são do tipo `char`, podemos escrever

```
char desempilha (void) {
    return pilha[--t];
}

void empilha (char y) {
    pilha[t++] = y;
}
```

Estamos supondo aqui que as variáveis `pilha` e `t` são globais, ou seja, foram definidas fora do código das funções. (Para completar o pacote, precisaríamos de mais três funções: uma que crie uma pilha, uma que verifique se a pilha está vazia e uma que verifique se a pilha está cheia.)².

Fonte: Feofiloff (2018, on-line)²



eu recomendo!



livro

C Completo e Total

Autor: Herbert Schildt

Editora: Makron

Ano: 1997

Sinopse: *C Completo e Total* está dividido em cinco partes, cada uma abordando um aspecto importante de C. Parte 1: apresenta uma discussão detalhada da linguagem C, incluindo palavras-chave, tipos de dados, operadores, funções, ponteiros, E/S, alocação dinâmica e muito mais. Parte 2: apresenta uma extensa descrição das funções de biblioteca por categoria. Abrange as funções definidas pelo padrão ANSI e muitas extensões comuns, como as chamadas de arquivos Unix, os gráficos e as funções de interface com o sistema operacional. Parte 3: mostra como aplicar C, concentrando-se em algoritmos úteis e aplicações interessantes da linguagem C. Parte 4: trata do ambiente de desenvolvimento C, incluindo eficiência, portabilidade, depuração e interface com o código *assembler*. Parte 5: desenvolve um interpretador C, com vários e diversificados exemplos que esclarecem cada conceito apresentado neste livro e que o diferenciam de qualquer outra obra de referência sobre C.





FUNÇÕES E ARQUIVOS

PROFESSORES

Dra. Gislaine Camila Lapasini Leal
Me. Pietro Martins de Oliveira

PLANO DE ESTUDO

A seguir, apresentam-se as aulas que você estudará nesta unidade: • Funções e escopo de variáveis • Passagem de parâmetros • Recursividade e arquivos • Exercícios resolvidos.

OBJETIVOS DE APRENDIZAGEM

- Conhecer as funções e o escopo de variáveis • Entender a passagem de parâmetros e seus desdobramentos: por valor, por referência e protótipo de funções • Estudar a recursividade e os arquivos • Observar os exercícios resolvidos.

INTRODUÇÃO



Prezado(a) estudante, chegamos à última unidade da disciplina de Algoritmos e Lógica de Programação II. Nesta unidade, você estudará o conceito de modularização por meio de funções.

Geralmente, no dia a dia de um programador, são encontrados problemas complexos e abrangentes. Para resolver esse tipo de problema, o primeiro passo consiste em decompô-lo em subproblemas para facilitar o processo de entendimento, análise e resolução. Em linguagem C, utilizamos as funções – blocos de instruções que realizam tarefas específicas.

Ao trabalhar com funções, surge a necessidade de entender o conceito de escopo de variáveis, compreender o que são variáveis locais e globais e como elas se relacionam com o programa. O escopo está relacionado à visibilidade de uma variável dentro do código-fonte do programa, sendo que uma variável local é aquela visível apenas dentro da função, e uma variável global é aquela que está acessível de qualquer parte do algoritmo.

Ainda no contexto de funções, estudaremos a passagem de parâmetros por valor e por referência e o conceito de recursividade. Em relação à passagem de parâmetros, veremos quando utilizar cada uma delas e qual é sua relação com os parâmetros reais e formais. Estudaremos funções recursivas, que são funções que fazem chamadas a si mesmas, como construí-las, quais suas vantagens e desvantagens. Além disso, conheceremos o conceito de arquivos e como manipulá-los, utilizando operações que possibilitem consultar, inserir, modificar e eliminar dados.

Ao final desta unidade, você saberá construir programas utilizando funções e poderá responder a questões do tipo: quando utilizar uma função? Quando declarar uma variável local ou global? Quando utilizar protótipo de uma função? Como deve ser realizada a passagem de parâmetros? Como acessar um arquivo? Como percorrer um arquivo?

Bons estudos!



1 FUNÇÕES E ESCOPO de Variáveis

Para solucionar problemas complexos e abrangentes, em geral, temos que dividir o problema em subproblemas mais simples e específicos e, com isso, dividimos sua complexidade e facilitamos o processo de resolução. Este processo de decomposição é denominado de refinamento sucessivo ou abordagem *top-down* (FORBELLONE; EBERSPACHER, 2005).

No particionamento dos problemas complexos, utilizamos subrotinas para resolver cada subproblema, permitindo a modularização. A linguagem C possibilita a modularização por meio de funções. Uma função é uma subrotina que tem como objetivo desviar a execução do programa principal para realizar uma tarefa específica e retornar um valor. São estruturas que possibilitam ao usuário separar seus programas em blocos (ASCENCIO; CAMPOS, 2010).

Um programa em C é um conjunto de funções que são executadas a partir da execução de uma função denominada `main`. Cada função pode conter declarações de variáveis, instruções, ou até mesmo, outras funções.

O objetivo de uma função é realizar alguma subtarefa específica, isto é, podemos escrever funções para a entrada, processamento e saída de dados.

A sintaxe de uma função é dada por:

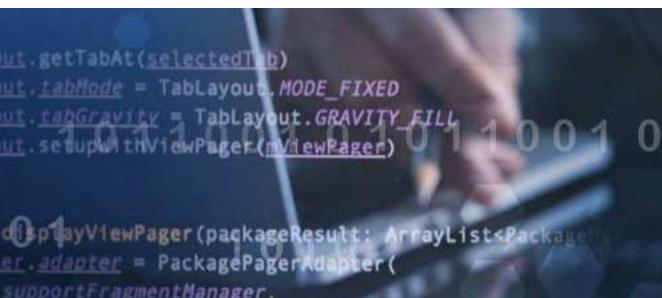
```
<tipo_de_retorno> <nome_da_função>(<parâmetros>)
{
<corpo_da_função>
}
```

em que:

- **tipo_de_retorno**: indica o tipo do dado que a função retornará.
- **parâmetros**: especificação das variáveis de entrada da função. Devemos especificar o tipo de cada uma das variáveis, podendo ter uma, várias ou nenhuma variável de entrada.
- **corpo_da_função**: conjunto de instruções que realizam a subtarefa, isto é, instruções que realizam o processamento dos dados de entrada e geram a saída de dados.

Em nossas funções, utilizamos o comando `return`, o qual é responsável por encerrar a função e retornar o conteúdo especificado por este comando. Lembre-se: o valor informado no comando `return` deve ser compatível com o tipo declarado para a função.

No Quadro 1, temos um programa que utiliza uma função denominada `soma` para efetuar leitura, processamento e saída de dados. A função foi declarada como `int` e não apresenta parâmetros de entrada. Na função principal (`main`), temos a chamada para a função `soma`, a qual realiza a entrada de dois valores numéricos, a `soma`, e exibe o resultado. Observe que as variáveis foram declaradas no interior da função `soma`, logo, estudaremos o escopo das variáveis – aí você entenderá a diferença entre declarar a variável dentro da função ou fora dela.



```
ut.getTabAt(selectedTab)
ut.tabMode = TabLayout.MODE_FIXED
ut.tabGravity = TabLayout.GRAVITY_FILL
ut.setupWithViewPager(viewPager)

displayViewPager(packageResult: ArrayList<Package
er.adapter = PackagePagerAdapter(
supportFragmentManager,
```

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int soma()
05 {
06     float num1, num2, total;
07     printf("Digite o primeiro número: ");
08     scanf("%f", &num1);
09     printf("Digite o segundo número: ");
10     scanf("%f", &num2);
11     total = num1 + num2;
12     printf("A soma é %.2f\n", total);
13     return (0);
14 }
15
16 int main()
17 {
18     system("cls");
19     soma();
20     return (0);
21 }
```

Quadro 1 - Programa em C / Fonte: os autores.

A linguagem C possui o tipo de dados **void**. Este tipo quer dizer vazio, em inglês, e nos permite escrever funções que não retornam nada e não possuem parâmetros. A sintaxe de uma função que não retorna nada é dada por algo como:

void <nome_da_função>(<parâmetros>)

Um exemplo utilizando o tipo **void** é apresentado no Quadro 2. Observe que, em funções do tipo **void**, não somos obrigados a utilizar o comando **return**, pois não há valor de retorno.

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 void soma()
05 {
06     float num1, num2, total;
```

```
07     printf("Digite o primeiro número: ");
08     scanf("%f", &num1);
09     printf("Digite o segundo número: ");
10     scanf("%f", &num2);
11     total = num1 + num2;
12     printf ("A soma é %.2f\n", total);
13 }
14
15 int main()
16 {
17     system("cls");
18     soma();
19     return (0);
20 }
```

Quadro 2 - Programa em C / Fonte: os autores.

Na seção seguinte, estudaremos o escopo das variáveis.

Escopo de variáveis

O escopo de uma variável é relacionado à sua visibilidade em relação às subrotinas de um programa. As variáveis declaradas no interior de uma função são chamadas de **variáveis locais**, pois podem ser utilizadas apenas dentro da função. Ao final da execução da função, essas variáveis são destruídas e seus conteúdos são perdidos (ASCENCIO; CAMPOS, 2010). As **variáveis globais** são aquelas declaradas fora das funções. Elas estão acessíveis em qualquer parte do programa e são destruídas apenas ao final da execução do programa (ASCENCIO; CAMPOS, 2010).

O que acontece se declararmos todas as variáveis como globais? Este tipo de variável ocupa memória durante toda a execução do programa e o torna mais difícil de ser entendido. Deste modo, devemos evitar ao máximo o seu uso.



PASSAGEM DE PARÂMETROS

As funções com passagem de parâmetros são aquelas que recebem valores no momento em que são chamadas. O Quadro 3 apresenta um programa utilizando a passagem de parâmetros.

A função `soma` é do tipo `float` e possui dois parâmetros `n1` e `n2`, declarados na linha 4, os quais também são do tipo `float`. Note que a entrada de dados é realizada nas linhas 16 e 18, dentro da função `main`. Após a leitura, é realizada uma chamada à função `soma`, **na linha 19**, passando, como argumentos, os conteúdos das variáveis `num1` e `num2`.

A função `soma` realiza a operação de adição (linha 7) dos dois valores obtidos como parâmetro e retorna o `total`. Como a função `soma` retorna um valor do tipo `float`, sua invocação foi atrelada a uma operação de atribuição, para que seu valor de retorno seja armazenado na variável `resposta` (linha 19).

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 float soma(float n1, float n2)
05 {
06     float total;
07     total = n1 + n2;
08     return (total);
```

```
09 }  
10  
11 int main()  
12 {  
13     float num1, num2, resposta;  
14     system("cls");  
15     printf("Digite o primeiro número: ");  
16     scanf("%f", &num1);  
17     printf("Digite o segundo número: ");  
18     scanf("%f", &num2);  
19     resposta = soma(num1, num2);  
20     printf("A soma é igual a %.2f\n", resposta);  
21 }
```

Quadro 3 - Programa em C / Fonte: os autores.

Na passagem de parâmetros, temos que os valores das variáveis num1 e num2 foram copiados para as variáveis n1 e n2, respectivamente. Na passagem de parâmetros, há distinção entre parâmetros reais e parâmetros formais. Nesse exemplo, os reais, ou argumentos, são os valores obtidos na entrada de dados, e os formais são os parâmetros declarados na definição da função. Com isso, temos que num1 e num2 são os parâmetros reais, e n1 e n2 são parâmetros formais.

A passagem de parâmetro ocorre quando é realizada a substituição dos parâmetros formais pelos reais no momento da execução da função. A passagem pode ser realizada de duas formas distintas: por valor ou por referência. A seguir, veremos cada uma delas.

Passagem de parâmetros por valor

A passagem de parâmetro **por valor** é caracterizada pela não alteração do conteúdo do parâmetro real, quando o parâmetro formal é manipulado na função. Isto é, qualquer alteração na variável local da função não afetará o valor do parâmetro real correspondente. Na passagem de parâmetros por valor, a função trabalha com cópias dos valores passados no momento de sua invocação (MANZANO; OLIVEIRA, 1997; ASCENCIO; CAMPOS, 2010).

No Quadro 4, é apresentado um programa em C que recebe um número inteiro, calcula o seu quadrado e exibe.

```
01 #include <stdio.h>
02
03 int calcQuadrado(int x)
04 {
05     x = x * x;
06     return x;
07 }
08
09 int main()
10 {
11     int num, resposta;
12     printf("Digite um número inteiro: ");
13     scanf("%d", &num);
14     resposta = calcQuadrado(num);
15     printf("O quadrado do número %d é %d\n", num,
16             resposta);
```

Quadro 4 - Programa em C / Fonte: os autores.

O parâmetro formal `x` da função `calcQuadrado` sofre alterações no interior da função (linha 5), mas a variável `num` da função `main` permanece inalterada, como pode ser visto na Figura 1.

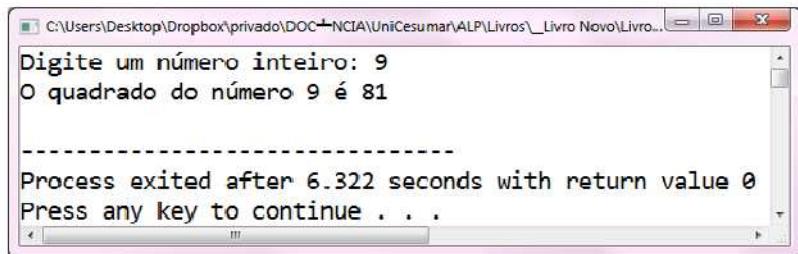


Figura 1 - Programa em C – Saída / Fonte: os autores.

Perceba que, na Figura 1, quando o usuário insere o valor 9, o parâmetro real continua valendo 9, mesmo depois da invocação da função.

Passagem de parâmetros por referência

Na passagem de parâmetro **por referência**, os parâmetros passados para a função consistem em endereços de memória ocupados por variáveis. O acesso a determinado valor é realizado por apontamento do endereço. Na passagem por referência, o valor do parâmetro real é alterado quando o parâmetro formal é manipulado dentro da função (ASCENCIO; CAMPOS, 2010).

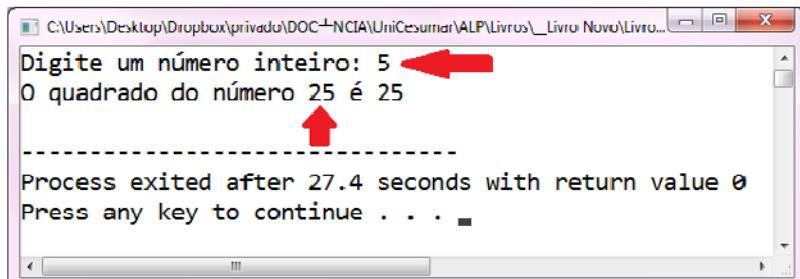
A passagem de parâmetros por referência é exemplificada no programa do quadro a seguir.

```
01 #include <stdio.h>
02
03 int calcQuadrado(int *x)
04 {
05     *x = *x * (*x);
06     return *x;
07 }
08
09 int main()
10 {
11     int num, resposta;
12     printf("Digite um número inteiro: ");
13     scanf("%d", &num);
14     resposta = calcQuadrado(&num);
15     printf("O quadrado do número %d é %d\n", num,
16         resposta);
```

Quadro 5 - Programa em C / Fonte: os autores.

Observe que, na chamada da função `calcQuadrado`, na linha 14, temos que passar o endereço de memória ocupado pela variável `num`. Isto é realizado pelo operador `&`, que obtém o endereço de memória de uma variável. Note, também, que as operações realizadas no interior da função são sobre ponteiros, pois, para que haja passagem por referência, é preciso que o parâmetro formal tenha de ser do tipo *pointer* (observe o asterisco “`*`” na declaração do parâmetro `x`, na linha 3). Assim, temos que inserir o operador “`*`” antes do identificador `x` (linhas 3, 5 e 6). Mas, ao final, o que muda em relação à execução?

A figura a seguir ilustra a saída obtida com a execução do programa. Podemos observar que, ao utilizar a passagem de parâmetros por referência, temos a alteração do conteúdo da variável num que, inicialmente, tinha o valor 5 e, ao final da execução da função calcQuadrado, ficou com o valor 25. Por que isso acontece?



```
C:\Users\Desktop\Dropbox\privado\DOC-LNCIA\UriCesumar\ALP\Livros\_Livro Novo\Livro...\n\nDigite um número inteiro: 5\nO quadrado do número 25 é 25\n-----\nProcess exited after 27.4 seconds with return value 0\nPress any key to continue . . .
```

Figura 2 - Programa em C – Saída / Fonte: os autores.

Porque não temos uma cópia do valor de num na passagem de parâmetros e, sim, o seu endereço de memória. Com isso, toda e qualquer alteração sobre o conteúdo da variável x também impactará o conteúdo da variável num, diretamente. É como se a variável x fosse a própria variável num.

Protótipo de funções

Você observou que, até o momento, todas as nossas funções foram definidas antes da função main? Mas será que podemos inserir nossas funções após a função main? Sim, em qualquer programa, podemos escrever as funções antes ou após a main. Se optarmos, no entanto, por escrevê-las depois da função main, temos que utilizar o protótipo de funções.

O protótipo de uma função é uma linha igual ao cabeçalho da função, acrescido de ponto e vírgula, que deve ser escrito antes da função main. Esta linha é responsável por informar ao compilador que outras funções deverão ser encontradas ao término do main (ASCENCIO; CAMPOS, 2010).

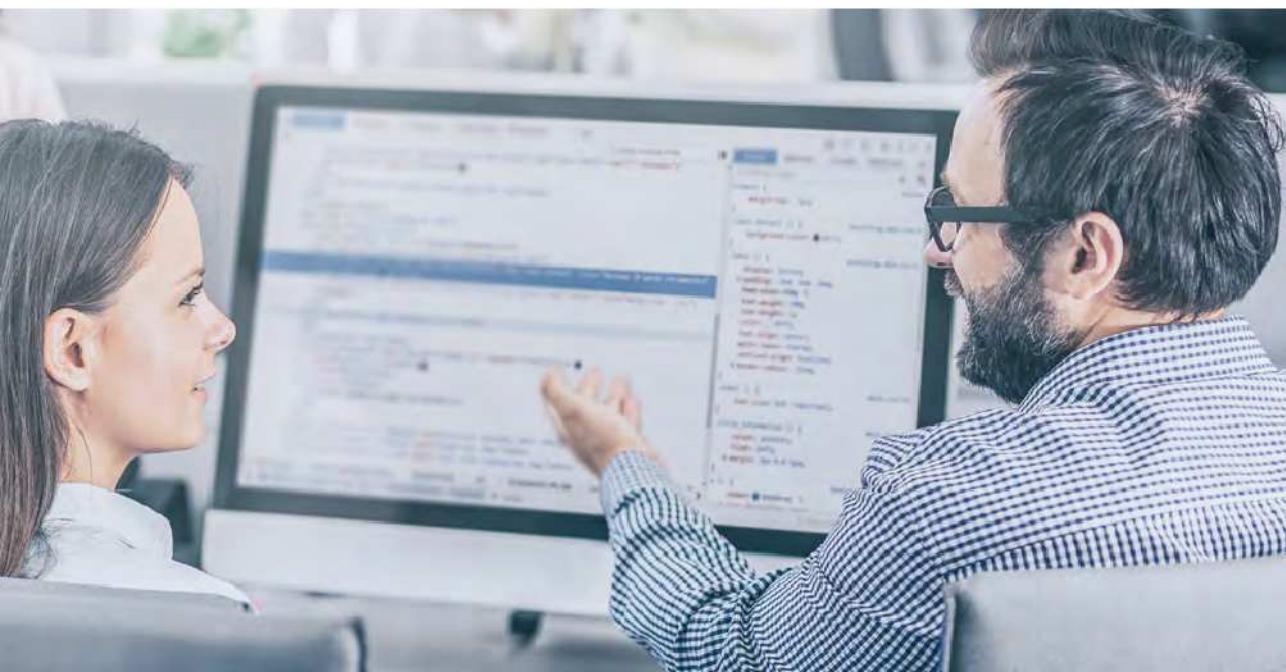
A sintaxe do protótipo de uma função é dada por:

```
<tipo_de_retorno> <nome_da_função>(<parâmetros>);
```

```
1 #include <stdio.h>
2
3 int calcQuadrado(int x);
4
5 int main()
6 {
7     int num, resposta;
8     printf("Digite um número inteiro: ");
9     scanf("%d", &num);
10    resposta = calcQuadrado(num);
11    printf("O quadrado do número %d é %d\n", num, resposta);
12 }
13
14 int calcQuadrado(int x)
15 {
16     x = x*x;
17     return x;
18 }
```

Figura 3 - Programa em C / Fonte: os autores.

O programa da Figura 3 apresenta um exemplo de programa com uma função escrita depois da função `main`. Nesse exemplo, para que o compilador reconheça a função `calcQuadrado`, temos que aplicar o conceito de protótipo de uma função, em que declaramos o cabeçalho da função `calcQuadrado` seguido de ponto e vírgula antes da função `main` (conforme indicado na linha 3 da Figura 3).



RECURSIVIDADE E ARQUIVOS

Do mesmo modo que, em outras linguagens, em C, é possível elaborar uma função que chama a si mesma, isto é, uma função recursiva. A recursividade é um mecanismo que permite uma função chamar a si mesma direta ou indiretamente. Uma função é recursiva quando possui uma chamada a si própria (ZIVIANE, 2004).

Ao construir funções recursivas, devemos nos certificar de que há um critério de parada, o qual determinará o momento que a função parará de fazer chamadas a si mesma, impedindo que entre em um *loop*.

No Quadro 6, temos a implementação da função recursiva para cálculo do fatorial utilizando a linguagem C.

```
01 #include <stdio.h>
02
03 int fatorial (int x)
04 {
05     if (x == 0)
06         return 1;
07     else
08         return x * fatorial(x-1);
09 }
10
11 int main()
12 {
```

```

13     int num, resposta;
14     printf("Digite um número inteiro: ");
15     scanf ("%d", &num);
16     resposta = fatorial(num);
17     printf ("O fatorial é %d\n", resposta);
18 }
```

Quadro 6 - Programa em C / Fonte: os autores.

O Quadro 7 apresenta a função recursiva para cálculo da série de Fibonacci.

```

01 #include <stdio.h>
02
03 int fibonacci (int x)
04 {
05     if ((x == 0) || (x==1))
06         return x;
07     else
08         return fibonacci(x-2) + fibonacci(x-1);
09 }
10
11 int main()
12 {
13     int num, resposta;
14     printf("Digite um número inteiro: ");
15     scanf ("%d", &num);
16     resposta = fibonacci(num);
17     printf ("O fibonacci é %d\n", resposta);
18 }
```

Quadro 7 - Programa em C / Fonte: os autores.

Arquivos

Os arquivos são utilizados no armazenamento de uma grande quantidade de informações por um grande período de tempo. Um arquivo pode ser lido ou escrito por um programa, sendo constituído por uma coleção de caracteres (arquivo texto) ou bytes (arquivo binário) (ASCENCIO; CAMPOS, 2010).

A biblioteca `stdio.h` oferece suporte à utilização de arquivos. Esta biblioteca fornece funções para manipulação de arquivos, define novos tipos de dados a serem usados, especificamente, com arquivos, como o tipo `FILE` (ASCENCIO; CAMPOS, 2010).

Uma variável do tipo ponteiro FILE identifica um arquivo no disco e direciona para ele todas as operações. A declaração deste tipo de variável é dada por algo como:

```
FILE *arq;
```

Para abrir um arquivo, utilizamos a função `fopen`, a qual abre um arquivo e retorna o ponteiro associado a ele. Considerando a variável FILE *arq, pode-se dizer que a sintaxe do `fopen` seria algo como:

```
arq = fopen (<nome_do_arquivo>, <modo_de_abertura>);
```

em que:

- arq: variável que armazena o endereço inicial de memória ocupado por um arquivo;
- nome_do_arquivo: nome do arquivo que se deseja abrir, podendo conter, ainda, o caminho absoluto para acessar esse arquivo;
- modo_da_abertura: representa como o arquivo será aberto.

r	Abre um arquivo de texto apenas para operações de leitura.
w	Cria um arquivo de texto em que pode ser realizada a escrita.
a	Anexa novos dados a um arquivo de texto.
r+	Abre um arquivo de texto em que podem ser realizadas leitura e escrita.
w+	Cria um arquivo de texto em que podem ser realizadas leitura e escrita.
a+	Anexa novos dados ou cria um arquivo para operações de leitura e escrita.

Quadro 8 - Modos para abertura de arquivos / Fonte: adaptado de Ascencio e Campos (2010).



explorando Ideias

A Linguagem C não permite que vetores e matrizes sejam passados, na íntegra, como parâmetros de uma função. Deve-se passar apenas o endereço da posição inicial – isso indica que um vetor ou matriz podem ser passados para uma função apenas por referência.

Fonte: Ascencio e Campos (2010).

A função **fopen**, quando utilizada no modo escrita, cria o arquivo especificado. Se este não existir ou se já existe um arquivo com o mesmo nome, será sobreposto por um novo arquivo vazio. O resultado da função **fopen** retorna o endereço de memória ocupado pelo arquivo, ou NULL, quando ocorre algum erro e o arquivo não é aberto (ROCHA, 2006; ASCENCIO; CAMPOS, 2010).

Para fechar um arquivo, utilizamos a função **fclose**, que possui a seguinte sintaxe:

```
fclose (arq) ;
```

Em que o argumento arq é a referência (variável ponteiro) para o arquivo. A função **fclose** retorna um valor inteiro. Um retorno igual a zero indica que o arquivo foi fechado corretamente.

A escrita de um caractere em um arquivo é realizada por meio da função **fputc**, que possui a seguinte sintaxe:

```
fputc (ch, arq) ;
```

em que:

- ch: corresponde ao char que será escrito no arquivo;
- arq: referência do arquivo em que o caractere será escrito.

A operação de leitura de um caractere é realizada utilizando a função **fgetc**. Sua sintaxe é:

```
fgetc (arq) ;
```

Se a execução desta função for bem-sucedida, o caractere será armazenado em uma variável do tipo **int** ou **char**.

A escrita de uma cadeia de caracteres é realizada por meio da função **fputs**, que apresenta a seguinte sintaxe:

```
fputs (cadeia, arq) ;
```

O argumento cadeia armazena a cadeia de caracteres que será escrita no arquivo, e o argumento arq é a referência para o arquivo em que a cadeia será escrita.

Na leitura de uma cadeia de caracteres, utilizamos a função **fgets**, cuja sintaxe é:

```
fgets(cadeia, tam, arq);
```

em que:

- **cadeia**: armazena a cadeia de caracteres obtida do arquivo, é um vetor de **char**;
- **tam**: indica que quantidade máxima de caracteres lidos será **tam-1**, um dado do tipo **int**;
- **arq**: referência para o arquivo.

Se quisermos gravar qualquer tipo de dado no arquivo, podemos utilizar a função **fwrite**. Sua sintaxe é dada por:

```
fwrite(mem, qtd_bytes, cont, arq);
```

em que:

- **mem**: representa a variável que armazena o conteúdo a ser gravado, podendo ser um dado de qualquer tipo;
- **qtd_bytes**: total de bytes a ser gravado no arquivo;
- **cont**: número de blocos de tamanho **qtd_bytes** que será armazenado;
- **arq**: referência para o arquivo.

Se a execução da função for bem-sucedida, seu retorno será igual ao valor de **cont**, isto é, o número de gravações realizadas. Se ocorrer algum erro, o retorno será menor que **cont**.

Para efetuar a leitura de qualquer tipo de dado de um arquivo, utilizamos a função **fread**, que apresenta a seguinte sintaxe:

```
fread(mem, qtd_bytes, cont, arq);
```

em que:

- **mem**: representa a variável que receberá o conteúdo lido, podendo ser de qualquer tipo;
- **qtd_bytes**: tamanho do bloco que será lido em bytes;

- `cont`: número de blocos que será lido;
- `arq`: referência para o arquivo.

De modo análogo à função `fwrite`, se sua execução for bem-sucedida, o retorno será igual ao número de leituras, isto é, o valor de `cont`. Senão o valor será menor que `cont`.

No Quadro 9, são apresentadas algumas funções para manipulação de arquivos.

Função	Sintaxe	Objetivo
<code>feof()</code>	<code>feof(arq);</code>	Verifica se o fim do arquivo foi atingido. Um valor de retorno igual a zero indica que o fim ainda não foi atingido.
<code>rewind()</code>	<code>rewind(arq);</code>	Posiciona o cursor no início do arquivo.
<code>remove()</code>	<code>remove(nomearq);</code>	Apaga um arquivo.
<code>rename()</code>	<code>rename(nome_atual, nome_novo)</code>	Altera o nome de um arquivo.

Quadro 9 - Funções para manipulação de arquivos / Fonte: os autores.

Fique tranquilo(a)! Veremos o funcionamento destas funções. O Quadro 10 apresenta um programa que exemplifica a manipulação de arquivos. É realizada a operação de abertura do arquivo, a verificação de erro, leitura de um caractere e o fechamento de arquivo. A leitura de caractere é realizada até que seja digitado o caractere ‘f’. Execute o código abaixo e, ao final, abra o arquivo com o nome “arquivo.txt” e verifique o seu conteúdo.

```

01 #include <stdio.h>
02
03 int main()
04 {
05     FILE *arq;
06     char letra;
07

```

```
08     arq = fopen("arquivo.txt", "w");
09     if (arq == NULL)
10     {
11         printf ("O arquivo não foi aberto. Ocorreu um
12             erro!\n");
13     }
14     else
15     {
16         printf ("Digite um caractere: ");
17         scanf("%c", &letra);
18         fflush(stdin);
19         while ((letra != 'f') && (letra !='F'))
20         {
21             fputc(letra, arq);
22             if (ferror(arq))
23             {
24                 printf("Erro na gravação !!!\n");
25             }
26             else
27             {
28                 printf ("Gravação efetuada com
29                     sucesso!\n");
30             }
31             printf ("Digite outro caractere: ");
32             scanf("%c", &letra);
33             fflush(stdin);
34         }
35     }
36 }
```

Quadro 10 - Programa em C / Fonte: os autores.

No Quadro 11, temos um exemplo de manipulação de arquivo com a gravação de cadeias de caracteres até que seja informada a palavra “fim”.

```
01 #include <stdio.h>
02 #include <string.h>
03
04 int main()
05 {
06     FILE *arq;
07     char palavra[50];
08
09     arq = fopen("arquivo.txt", "w");
10     if (arq == NULL)
11     {
12         printf ("O arquivo não foi aberto. Ocorreu um
13             erro!\n");
14     }
15     else
16     {
17         printf ("Digite uma palavra: ");
18         gets(palavra);
19         fflush(stdin);
20         while ((strcmp(palavra,"fim") != 0))
21         {
22             fputs(palavra, arq);
23             if (ferror(arq))
24             {
25                 printf("Erro na gravação!!!\n");
26             }
27             else
28             {
29                 printf ("Gravação efetuada com
30                     sucesso!\n");
31             }
32             printf ("Digite uma palavra: ");
33             gets(palavra);
34             fflush(stdin);
35         }
36     }
37 }
```

Quadro 11 - Programa em C / Fonte: os autores.

Um exemplo de operação de leitura de um arquivo - texto é apresentado no Quadro 12. A leitura do arquivo é realizada até que seja encontrado o fim dele, de modo que são efetuadas leituras com até 50 caracteres.

```
01 #include <stdio.h>
02 #include <string.h>
03 int main()
04 {
05     FILE *arq;
06     char frase[50];
07     arq = fopen("arquivo.txt", "r");
08     if (arq == NULL)
09     {
10         printf ("O arquivo não foi aberto. Ocorreu um
11             erro!\n");
12     }
13     else
14     {
15         while (!feof(arq))
16         {
17             fgets(frase, 50, arq);
18             if (ferror(arq))
19             {
20                 printf("Erro na leitura do arquivo!!!\n");
21             }
22             else
23             {
24                 printf("Leitura realizada com sucesso. A
25                     cadeia é: %s \n", frase);
26             }
27         }
28     }
29 }
```

Quadro 12 - Programa em C / Fonte: os autores.

Observe que, quando queremos realizar a leitura ou a escrita de um arquivo, devemos utilizar as funções de abertura, verificação de erro e fechamento. Em relação à abertura do arquivo, não podemos esquecer de modificar o modo de

acesso de acordo com o que desejamos realizar. Nos programas do Quadro 10 e do Quadro 11, a abertura do arquivo foi realizada utilizando o modo de acesso de gravação (w), e no Quadro 12, foi realizada utilizando o modo de acesso apenas de leitura (r). Podemos abrir um arquivo para realizar leitura e escrita? Sim, basta identificar o modo de acesso como r+. Para relembrar os modos de acesso a arquivos, verifique o Quadro 8.

AULA

EXERCÍCIOS RESOLVIDOS

Problema 1

Escreva um programa utilizando uma função que converta dada temperatura lida de Celsius para Fahrenheit.

```
01 #include <stdio.h>
02
03 float convertet(float celsius);
04
05 int main()
```

```
06  {
07      float celsius, resposta;
08      printf("Informe a temperatura em graus Celsius: ");
09      scanf("%f", &celsius);
10      resposta = convertet(celsius);
11      printf("A temperatura %.2f em Fahrenheit é %.2f\n",
12          celsius, resposta);
13  }
14  float convertet(float celsius)
15  {
16      float temp;
17      temp = celsius * 1.8 + 32;
18      return temp;
19 }
```

Quadro 13 - Programa em C, Problema 1 / Fonte: os autores.

Problema 2

Escreva um programa utilizando uma função que receba o peso de um peso em quilogramas e o converta para libras.

```
01  #include <stdio.h>
02
03  float convertep (float peso);
04
05  int main()
06  {
07      float peso, resposta;
08      printf("Informe o peso em quilogramas: ");
09      scanf ("%f", &peso);
10      resposta = convertep(peso);
11      printf ("O peso %.2f em libras é %.2f\n", peso,
12          resposta);
```

```
12 }
13
14 float convertep (float peso)
15 {
16     peso = peso * 2.68;
17     return peso;
18 }
```

Quadro 14 - Programa em C, Problema 2 / Fonte: os autores.

Problema 3

Faça uma função que receba, como parâmetro, um vetor com dez números inteiros e retorne-os ordenados em forma crescente.

```
01 #include <stdio.h>
02
03 float convertep (float peso);
04
05 int main()
06 {
07     float peso, resposta;
08     printf("Informe o peso em quilogramas: ");
09     scanf ("%f", &peso);
10     resposta = convertep(peso);
11     printf ("O peso %.2f em libras é %.2f\n", peso,
12             resposta);
13
14 float convertep (float peso)
15 {
16     peso = peso * 2.68;
17     return peso;
18 }
```

Quadro 15 - Programa em C, Problema 3 / Fonte: os autores.

Problema 4

Elabore uma função que receba uma *string* e retorne a quantidade de consoantes.

```
01 #include <stdio.h>
02 #include <string.h>
03
04 int conta(char nome[])
05 {
06     int i, tam, qtd;
07
08     tam = strlen(nome);
09     nome = strupr(nome);
10     qtd = 0;
11     for (i=0;i<tam;i++)
12     {
13         if ((nome[i] != 'A') && (nome[i] !='E') &&
14             (nome[i] !='I') && (nome[i] !='O') &&
15             (nome[i] != 'U') && (nome[i] != ' '))
16             qtd++;
17     }
18
19     return qtd;
20 }
21
22 int main()
23 {
24     int total;
25     char palavra[30];
26     printf("Informe a string: ");
27     gets(palavra);
28     total = conta(palavra);
29     printf("A quantidade de consoantes da string %s é
30         %d\n", palavra, total);
31 }
```

Quadro 16 - Programa em C, Problema 4 / Fonte: os autores



Para saber um pouco mais sobre a construção de funções e a passagem de parâmetros, acesse:

http://pt.wikibooks.org/wiki/Programar_em_C/Fun%C3%A7%C3%B5es.

Fonte: os autores.



CONSIDERAÇÕES FINAIS

Nesta unidade, você aprendeu como modularizar os programas utilizando funções. Vimos que as funções são subrotinas que nos permitem desviar a execução do programa principal para realizar uma tarefa específica e, assim, retornam um valor. Além disso, estudamos que um programa em C é um conjunto de funções que são executadas a partir da execução da função denominada **main()**, e que cada função pode conter declarações de variáveis, instruções, ou até mesmo, outras funções.

Ainda em relação a funções, vimos o uso do comando **return**, responsável por encerrar a função e retornar o dado indicado. Lembre-se de que o dado retornado deve ser compatível com o tipo da função declarada. Conhecemos o conceito do escopo de variáveis locais e globais, sendo que uma variável local é aquela que está acessível apenas dentro da função, enquanto que uma variável global é acessível de qualquer parte do programa.

Estudamos a passagem de parâmetros por valor e por referência. Na passagem de parâmetros por valor, não há alteração do conteúdo do parâmetro real, pois a função trabalha com cópias dos valores passados, no momento de sua chamada. Já na passagem de parâmetros por referência, estes conteúdos podem ser alterados.

Abordamos o conceito de protótipo de uma função, que informa ao compilador que outras funções deverão ser encontradas ao término do **main**. Este conceito nos permite escrever funções depois da função **main**.

Estudamos o conceito de recursividade e construímos programas para as funções recursivas de cálculo de fatorial e série de Fibonacci.

Por fim, trabalhamos com a manipulação de arquivos, realizando operações de leitura e escrita. Aprendemos como abrir arquivos, quais os modos de operação, como verificar erros durante a abertura, como realizar leitura e escrita de caractere e de cadeia de caracteres e construímos programas para colocar em prática os conceitos vistos.



1. É comum, em uma aplicação, ter de determinar quais números são pares ou ímpares entre todos os valores de um conjunto de dados. Dessa forma, faça um programa que verifica se determinado número é positivo ou negativo, por meio de uma função.
2. Provavelmente, você já deve ter se deparado com uma situação na qual é preciso calcular o somatório de valores compreendidos dentro de um intervalo determinado. Por isso, elabore uma função que receba dois números positivos por parâmetro e retorne a soma dos n números inteiros existentes entre eles.
3. Imagine que você está desenvolvendo uma pequena funcionalidade dentro de um programa que será utilizado como processador de texto. Assim sendo, escreva uma função que receba um caractere e retorne zero se for vogal ou 1 se for uma consoante, um número ou caractere especial.
4. É muito comum que programas tenham de implementar funcionalidades para identificar características específicas e, assim, realizar operações sobre um conjunto de dados. Dessa forma, faça um programa com uma função que apresente o somatório dos n primeiros números pares, definidos por um operador. O valor de n será informado pelo usuário.



5. No jardim de infância, durante o processo de ensino e aprendizagem, um professor costuma ensinar o tema “vogais e consoantes” de maneira bastante lúdica. Para auxiliar o professor, construa uma função que receba um nome e retorne o número de vogais.
6. O mundo das finanças é fascinante. Dentro deste contexto, existem diversas casas de câmbio que realizam compra e venda de moedas estrangeiras. Desse modo, elabore um programa que receba o valor da cotação do dólar e o valor em reais e que apresente o valor em dólares.
7. Os arquivos são recursos que nos permitem realizar o armazenamento persistente de dados, além de armazenar grande volume de informações. Dessa forma, construa um programa que permita ao usuário gravar dez palavras em um arquivo e, em seguida, efetue a leitura do arquivo e apresente o conteúdo na tela.



ARQUIVOS DE CABEÇALHO

Um arquivo de cabeçalho é um arquivo com extensão .h que contém declarações de funções em linguagem C e definições de macros a serem compartilhadas entre vários arquivos - fonte. Existem dois tipos de arquivos de cabeçalho: os arquivos que o programador cria e os arquivos que acompanham o seu compilador desde a sua instalação.

Você pode solicitar o uso de um arquivo de cabeçalho no seu programa, incluindo-o por meio de diretiva de pré-processamento C #include, assim como você viu em códigos - fonte anteriores, a inclusão do arquivo de cabeçalho stdio.h, que acompanha o seu compilador.

A inclusão de um arquivo de cabeçalho é igual a copiar o conteúdo do arquivo de cabeçalho, mas não o fazemos porque isso pode causar erros e não é uma boa ideia copiar o conteúdo de um arquivo de cabeçalho nos arquivos - fonte, especialmente, se temos vários arquivos - fonte em um programa ou projeto.

Uma prática simples nos programas C ou C ++ é que mantemos todas as constantes, macros, variáveis globais em todo o sistema, e protótipos de funções nos arquivos de cabeçalho, e incluímos esse arquivo de cabeçalho sempre que necessário.

Os arquivos de cabeçalho do usuário e do sistema são incluídos usando a diretiva de pré-processamento #include. Ele tem as duas formas a seguir:

```
#include <arquivo>
```

Esta forma é usada para arquivos de cabeçalho do sistema. Ele procura por um arquivo chamado 'arquivo' em uma lista padrão de diretórios do sistema. Você pode anexar diretórios a esta lista com a opção -I enquanto compila seu código-fonte.

```
#include "arquivo"
```

Esta forma é usada para arquivos de cabeçalho do seu próprio programa. Ele procura por um arquivo chamado 'arquivo' no diretório que contém



o arquivo atual. Você pode anexar diretórios a esta lista com a opção `-I` enquanto compila seu código-fonte.

A diretiva `#include` funciona direcionando o pré-processador C para varrer o arquivo especificado como entrada antes de continuar com o restante do arquivo fonte atual. A saída do pré-processador contém a saída já gerada, seguida pela saída resultante do arquivo incluído, seguida pela saída que vem do texto após a diretiva `#include`. Por exemplo, se você tiver um arquivo de cabeçalho intitulado `cabecalho.h` com o seguinte conteúdo (código):

```
char *teste(void);
```

e um programa principal chamado `programa.c` que usa o arquivo de cabeçalho, como o que se segue:

```
int x;
#include "cabecalho.h"
int main(void){
    puts(teste());
}
```

o compilador verá o mesmo conteúdo de código-fonte que faria se o `programa.c` fosse lido da seguinte forma:

```
int x;
char *teste(void);
int main(void){
    puts(teste());
}
```

Fonte: adaptado de Tutorials Point ([2020], on-line)³.



eu recomendo!



livro

Lógica de Programação – A Construção de Algoritmos e Estruturas de Dados - 3ª Edição

Autores: André Luiz Villar Forbellone e Henri F. Eberspacher

Editora: Makron Books

Ano: 2005

Sinopse: este livro introduz o leitor ao universo da lógica aplicada à programação de computadores. Ao final do estudo, o aluno estará capacitado a construir algoritmos, assim como a assimilar, mais facilmente, qualquer linguagem de programação existente ou futura. O texto não requer nenhum conhecimento prévio de informática e é independente de características de máquina. Cada capítulo conta com exercícios de fixação, que visam a sedimentar os assuntos de cada subitem, e com exercícios propostos, que cobrem todo o conteúdo do capítulo. No anexo, encontram-se resoluções dos exercícios de fixação. A pseudolínguagem utilizada é intencionalmente próxima das linguagens de programação comumente adotadas como primeira linguagem, para facilitar a posterior tradução e a implementação prática.





Caro(a) aluno(a),

Chegamos ao final de nossa disciplina de Algoritmos e Lógica de Programação II, em que você aprendeu os conceitos básicos da Linguagem de Programação C.

Na Unidade 1, vimos o histórico da Linguagem C, suas características e como o código de um programa é convertido em um arquivo executável. Aprendemos que todo programa possui uma função *main*, que é chamada quando o programa é executado, também vimos como inserir comentários em nossos códigos e as regras para a nomeação dos identificadores. Conhecemos os tipos básicos de dados, operadores e funções intrínsecas disponíveis na linguagem C. Trabalhamos o modo de realizar atribuição, entrada e saída de dados.

A Unidade 2 abordou a construção de programas utilizando a estrutura condicional, isto é, programas com desvio de fluxo. Estudamos a estrutura condicional simples, a condicional composta e a estrutura case. Aprendemos que a estrutura condicional é utilizada em situações em que um conjunto de instruções deve ser executado apenas se uma condição for verdadeira.

Na Unidade 3, vimos como construir algoritmos utilizando estruturas de repetição, que permitem a execução de um trecho de código repetidas vezes.

A Unidade 4 apresentou os conceitos relacionados a vetores, *strings*, matrizes e estruturas. Estudamos que os vetores e as matrizes são estruturas de dados homogêneas que agrupam diversas informações, do mesmo tipo, em uma única variável, e o acesso a essas estruturas é indexado. Trabalhamos a definição de novos tipos de dados utilizando estruturas (*structs*), que são capazes de armazenar informações de tipos diferentes.

Por fim, na Unidade 5, aprendemos a construir programas modulares com o uso de funções. Estudamos o conceito de escopo de variáveis e a passagem de parâmetros por valor e por referência. Abordamos a construção de programas utilizando protótipo de uma função, recursividade e manipulação de arquivos. Nestas cinco unidades, consolidamos o aprendizado dos conceitos iniciais da Linguagem de Programação C.

Após a explanação desses conceitos, sugerimos que você não pare por aqui e continue em busca do conhecimento. Muito sucesso a você.

ASCENCIO, A. F. G.; CAMPOS, E. A. V. **Fundamentos da programação de computadores**. 5. ed. São Paulo: Prentice Hall, 2010.

FORBELLONE, A. L. V.; EBERSPACHER, H. F. **Lógica de Programação**. 3. ed. São Paulo: Makron Books, 2005.

KERNIGHAN, B. W.; RITCHIE, D. M. **C Programming Language**. 2. ed. [S.I.]: Prentice Hall Software Series, 1988.

LOPES, A.; GARCIA, G. **Introdução à Programação**. Rio de Janeiro: Elsevier, 2002.

MANZANO, J. A. N. G.; OLIVEIRA, J. F. **Estudo dirigido de algoritmos**. 3. ed. São Paulo: Érica, 1997.

MENDES, R. D. Inteligência artificial: sistemas especialistas no gerenciamento da informação. **Ciência da Informação**. Brasília, v. 26, n. 1, jan./abr. 1997. Disponível em: http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0100-19651997000100006. Acesso em: 7 jan. 2020.

PAPPAS, C. H.; MURRAY, W. H. **Turbo C++ Completo e Total**. São Paulo: Makron: McGraw-Hill, 1991.

ROCHA, A. A. **Introdução à Programação Usando C**. Lisboa: FCA, 2006.

WIRTH, N. **Algoritmos e Estruturas de Dados**. Rio de Janeiro: LTC, 1999.

ZIVIANE, N. **Projeto de Algoritmos com implementações em Pascal e C**. 2. ed. São Paulo: Pioneira Thomson Learning, 2004.

REFERÊNCIAS ON-LINE

¹ Em: <http://cs.uno.edu/~jaime/Courses/2025/devCpp2025Instructions.html>. Acesso em: 3 dez. 2019.

² Em: <https://www.ime.usp.br/~pf/algoritmos/aulas/pilha.html>. Acesso em: 10 jan. 2020.

³ Em https://www.tutorialspoint.com/cprogramming/c_header_files.htm. Acesso em: 13 jan. 2020.

UNIDADE 1

1. A.

2.

```
01 #include <stdio.h>
02 int main()
03 {
04     char nome[20]; /*Nome com até 20 letras*/
05     printf("\n Digite seu nome: ");
06     scanf("%s", nome);
07     printf("\n Bem-vindo à disciplina de Algoritmos e
Lógica de Programação II %s", nome);
08     return (0);
09 }
```

3.

```
01 #include <stdio.h>
02 #include <math.h>
03 int main()
04 {
05     int valor;
06     printf("Digite um número inteiro: ");
07     scanf("%d", &valor);
08     printf("\n A potência de %d é %.2f", valor,
pow(valor,2));
09     printf("\n A potência de %d é %.2f\n", valor,
sqrt(valor));
10     return (0);
11 }
```

4.

```
01 #include <stdio.h>
02 int main()
03 {
04     float n1, n2, n3, n4, media;
05     printf("\n Digite a nota 1: ");
06     scanf("%f", &n1);
07     printf("\n Digite a nota 2: ");
08     scanf("%f", &n2);
09     printf("\n Digite a nota 3: ");
10     scanf("%f", &n3);
11     printf("\n Digite a nota 4: ");
12     scanf("%f", &n4);
13     media = (n1 + n2 + n3 + n4)/4;
14     printf("\n A média é: %.2f", media);
15     return (0);
16 }
```

5.

```
01 #include <stdio.h>
02 #include <math.h>
03 #define pi 3.141593
04 int main()
05 {
06     float r; // r (raio da circunferência)
07     printf(" Digite o valor do raio: ");
08     scanf("%f", &r);
09     printf("\n Valor da Área: %f", pi * pow(r,2));
10     printf("\n Valor da Perímetro: %f", 2 * pi * r);
11     return (0);
12 }
```

6.

```
01  /* insere o conteúdo do arquivo stdio.h */
02  #include <stdio.h>
03
04  int main()
05  { /* declaração das variáveis */
06      int num1, num2, total;
07      /*entrada de dados */
08      /*mensagem ao usuário */
09      printf("\nDigite o primeiro número: ");
10      /* leitura do primeiro número */
11      scanf("%d", &num1);
12      /*mensagem ao usuário */
13      printf("\nDigite o segundo número:");
14      /* leitura do segundo número*/
15      scanf("%d", &num2);
16      /* processamento */
17      /*cálculo do ano de nascimento */
18      total = num1 + num2;
19      /*saída de dados */
20      printf ("\n A soma dos números é: %d ", total);
21      return (0);
22 }
```

UNIDADE 2

1.

```
01 #include <stdio.h>
02 int main()
03 {
04     int num;
05     int maior, menor, i;
06
07     printf ("Digite o 1º número: \n");
08     scanf ("%d", &num);
09     maior = num;
10     menor = num;
11
12     printf ("Digite o 2º número: \n");
13     scanf ("%d", &num);
14     if (num > maior)
15         maior = num;
16     if (num < menor)
17         menor = num;
18
19     printf ("Digite o 3º número: \n");
20     scanf ("%d", &num);
21     if (num > maior)
22         maior = num;
23     if (num < menor)
24         menor = num;
25
26     printf ("Digite o 4º número: \n");
27     scanf ("%d", &num);
28     if (num > maior)
29         maior = num;
```

```
30     if (num < menor)
31         menor = num;
32
33     printf ("Digite o 5º número: \n");
34     scanf ("%d", &num);
35     if (num > maior)
36         maior = num;
37     if (num < menor)
38         menor = num;
39
40     printf("O maior é %d\n", maior);
41     printf("O menor é %d\n", menor);
42     return (0);
43 }
```

2.

```
01 #include <stdio.h>
02 #include <math.h>
03 int main()
04 {
05     int numero;
06     printf("Digite o número: \n");
07     scanf("%d", &numero);
08     if (numero % 3 == 0)
09         printf ("O número é divisível por 3.\n");
10     if (numero % 7 == 0)
11         printf ("O número é divisível por 7.\n");
12     return (0);
13 }
```

3.

```
01 #include <stdio.h>
02 int main()
03 {
04     int numero;
05     printf("Digite o número: ");
06     scanf("%d", &numero);
07     switch (numero)
08     {
09         case 1: printf("\nJaneiro\n");
10         break;
11         case 2: printf("\nFevereiro\n");
12         break;
13         case 3: printf("\nMarco\n");
14         break;
15         case 4: printf("\nAbril\n");
16         break;
17         case 5: printf("\nMaio\n");
18         break;
19         case 6: printf("\nJunho\n");
20         break;
21         case 7: printf("\nJulho\n");
22         break;
23         case 8: printf("\nAgosto\n");
24         break;
25         case 9: printf("\nSetembro\n");
26         break;
27         case 10: printf("\nOutubro\n");
28         break;
29         case 11: printf("\nNovembro\n");
30         break;
31         case 12: printf("\nDezembro\n");
32         break;
33         default: printf("\nMês inválido\n");
34         break;
35     }
36     return (0);
37 }
```

4.

```
01 #include <stdio.h>
02 int main()
03 {
04     int cargo;
05     float salario, aumento, salario_final;
06     printf("Digite o número do cargo do funcionário:
07         \n");
08     scanf("%d", &cargo);
09     printf("Digite o valor do salário do funcionário:
10         \n");
11     scanf("%f", &salario);
12     switch (cargo) {
13         case 1:
14             aumento = salario * 0.4;
15             salario_final = salario + aumento;
16             printf("O Servente teve aumento de R$%.2f e
17                 agora recebe R$%.2f\n", aumento,
18                 salario_final);
19             break;
20         case 2:
21             aumento = salario * 0.35;
22             salario_final = salario + aumento;
23             printf("O Pedreiro teve aumento de R$%.2f e
24                 agora recebe R$%.2f\n", aumento,
25                 salario_final);
26             break;
27         case 3:
28             aumento = salario * 0.20;
29             salario_final = salario + aumento;
30             printf("O Mestre de Obras teve aumento de
31                 R$%.2f e agora recebe R$%.2f\n", aumento,
32                 salario_final);
33             break;
34         case 4:
```

```
27     aumento = salario * 0.10;
28     salario_final = salario + aumento;
29     printf("O Técnico de Segurança teve aumento de
30           R$%.2f e agora recebe R$%.2f\n", aumento,
31           salario_final);
32     break;
33 }
```

5.

```
01 #include <stdio.h>
02 int main()
03 {
04     int cod_Estado, peso, cod_Carga;
05     float imposto, taxa_imp, preco, preco_quilo;
06     float total;
07
08     printf("Digite o código do Estado: \n");
09     scanf("%d", &cod_Estado);
10     printf("Digite o peso da carga em toneladas:
11           \n");
12     scanf("%d", &peso);
13     printf("Digite o código da carga: \n");
14     scanf("%d", &cod_Carga);
15
16     switch (cod_Estado) {
17         case 1:
18             taxa_imp = 0.2;
19             break;
20         case 2:
21             taxa_imp = 0.15;
22             break;
23         case 3:
```

```
23         taxa_imp = 0.1;
24         break;
25     case 4:
26         taxa_imp = 0.05;
27         break;
28     default:
29         taxa_imp = 0;
30         printf ("\nCódigo de estado inválido\n");
31         break;
32     }
33     if (cod_Carga >= 10 && cod_Carga <= 20)
34     {
35         preco_quilo = 180;
36     }
37     else if (cod_Carga >= 21 && cod_Carga <= 30)
38     {
39         preco_quilo = 120;
40     }
41     else if (cod_Carga >= 31 && cod_Carga <= 40)
42     {
43         preco_quilo = 230;
44     }
45     else{
46         printf ("\nCódigo de carga inválido\n");
47         preco_quilo = 0;
48     }
49
50     imposto = peso * 1000 * preco_quilo * taxa_imp;
51     preco = peso * 1000 * preco_quilo;
52     total = preco + imposto;
53
54     printf ("\nPeso em kg: %d\nPreco: %.2f\nImposto:
55     %.2f\nTotal: %.2f\n", peso*1000, preco, imposto,
total);
```

UNIDADE 3

1.

```
01 #include <stdio.h>
02 #include <math.h>
03 int main()
04 {
05     int numero, soma, quantidade, maior, menor, pares;
06     int impares;
07     float media;
08     numero = 1;
09     quantidade = 0;
10     soma = 0;
11     maior = 0;
12     menor = HUGE_VAL;
13     pares = 0;
14     impares = 0;
15     while (numero != 0)
16     {
17         printf("Digite um número:\n");
18         scanf("%d", &numero);
19         if (numero != 0)
20             quantidade++;
21         soma = soma + numero;
22         if (numero % 2 == 0 && numero != 0)
23             pares++;
24         else if (numero % 2 == 1 && numero != 0)
25             impares++;
26         if (numero > maior && numero != 0)
27             maior = numero;
28         if (numero < menor && numero != 0)
29             menor = numero;
30     }
31     media = soma / quantidade;
32     printf ("\nMédia: %.2f \nMaior: %d \nMenor: %d
33     \nPares: %d \nÍmpares: %d\n", media, maior, menor,
34     pares, impares);
35 }
```

2.

```
01 #include <stdio.h>
02 int main()
03 {
04     int numero, i;
05     float resposta, divisor;
06     resposta = 0;
07     printf("Digite um número:\n");
08     scanf("%d", &numero);
09     for (i=1; i <= numero; i++)
10     {
11         divisor = i;
12         resposta = resposta + 1/divisor;
13     }
14     printf("Resposta: %f\n", resposta);
15     return (0);
16 }
```

3.

```
01 #include <stdio.h>
02 int main()
03 {
04     int valor, i, j;
05     for (i = 1; i <= 10; i++)
06     {
07         printf ("Tabuada do %d\n", i);
08         for (j = 1; j <= 10; j++)
09         {
10             valor = i*j;
11             printf ("%d x %d = %d\n", i, j, valor);
12         }
13     }
14     return (0);
15 }
```

4.

```
01 #include <stdio.h>
02 int main()
03 {
04     float soma;
05     int i;
06     for (i = 200; i <= 500; i++)
07     {
08         if (i % 2 == 1)
09             soma = soma + i;
10     }
11     printf("A soma de todos os ímpares inteiros entre
12         200 e 500 é %.0f\n", soma);
13 }
```

5.

```
01 #include <stdio.h>
02 int main()
03 {
04     int i;
05     for (i = 1; i <= 30; i++)
06     {
07         if (i % 3 == 0)
08             printf ("O número %d é divisível por 3\n", i);
09         if (i % 7 == 0)
10             printf ("O número %d é divisível por 7\n", i);
11     }
12     return (0);
13 }
```

6.

```
01 #include <stdio.h>
02 int main()
03 {
04     char frase[30];
05     int i, numero;
06     printf("Digite a frase:\n");
07     //para poder digitar a frase com espaços em branco
08     gets(frase);
09     printf("Digite o número de repetições:\n");
10     scanf("%d", &numero);
11     for (i = 1; i <= numero; i++)
12     {
13         printf("%s\n", frase);
14     }
15     return (0);
16 }
```

7.

```
01 #include <stdio.h>
02 int main()
03 {
04     int pedido, dia, mes, ano, quantidade;
05     float preco_Unit, preco_Total;
06     pedido = 1; preco_Total = 0;// inicializacao
07     while (pedido != 0)
08     {
09         printf("Digite o numero do pedido: ");
10         scanf("%d", &pedido);
11         if (pedido != 0)
12         {
13             printf("Digite o dia do pedido:\n");
14             scanf("%d", &dia);
15             printf("\nDigite o mes do pedido:\n");
16             scanf("%d", &mes);
```

```
17         printf("\nDigite o ano do pedido:\n");
18         scanf("%d", &ano);
19         printf("\nDigite o preço unitário:\n");
20         scanf("%f", &preco_Unit);
21         printf("\nDigite a quantidade do
22         pedido:\n");
23         scanf("%d", &quantidade);
24         preco_Total = preco_Total + preco_Unit *
25         quantidade;
26     }
27     printf ("O preço total dos Pedidos é: %.2f\n",
28         preco_Total);
29     return (0);
30 }
```

8.

```
01 #include <stdio.h>
02 int main()
03 {
04     int idade, sexo, est_Civil, q_Casadas, quantidade;
05     int q_Solteiras, q_Separadas, q_Viuvas;
06     float peso, t_Peso, m_Peso, t_Idade, m_Idade;
07     idade = 1; t_Peso = 0; t_Idade = 0;
08     q_Casadas = 0; q_Solteiras = 0; q_Separadas = 0;
09     q_Viuvas = 0; quantidade = 0; //
10     while (idade != 0)
11     {
12         printf("Digite a idade:\n");
13         scanf("%d", &idade);
14         if (idade != 0)
15         {
16             quantidade++;
```

```
17         t_Idade = t_Idade + idade;
18         printf("Digite o peso:\n");
19         scanf("%f", &peso);
20         t_Peso = t_Peso + peso;
21         printf("Digite o sexo (1-M 2-F):\n");
22         scanf("%d", &sexo);
23         printf("Digite o estado civil (1-Casadas
24             2-Solteiras 3-Separadas 4-Viúvas):\n");
25         scanf("%d", &est_Civil);
26         switch (est_Civil)
27         {
28             case 1: q_Casadas++;
29                 break;
30             case 2: q_Solteiras++;
31                 break;
32             case 3: q_Separadas++;
33                 break;
34             case 4: q_Viuvas++;
35                 break;
36             default: printf("Estado Civil
37                         Inválido\n");
38         }
39     }
40     m_Idade = t_Idade / quantidade;
41     m_Peso = t_Peso / quantidade;
42     printf("\nCasadas: %d \nSolteiras: %d \nSeparadas:
43             %d \nViúvas: %d", q_Casadas, q_Solteiras,
44             q_Separadas, q_Viuvas);
45     printf("\nMédia de Peso: %.3f \nMédia de Idade:
46             %.1f\n", m_Peso, m_Idade);
47     return (0);
48 }
```

9.

```
01 #include <stdio.h>
02 #include <string.h>
03 int main()
04 {
05     char comodo[20] = "";
06     int largura, comprimento, area, i;
07     area = 0; // inicializacao
08     while (strcmp(comodo, "fim") != 0 &&
09           strcmp(comodo, "FIM") != 0)
10     {
11         printf("Digite o nome do cômodo:\n");
12         scanf("%s", &comodo);
13         if (strcmp(comodo, "fim") != 0 &&
14             strcmp(comodo, "FIM") != 0)
15         {
16             printf("Digite a largura do cômodo:\n");
17             scanf("%d", &largura);
18             printf("\nDigite o comprimento do
19                   cômodo:\n");
20             scanf("%d", &comprimento);
21             area = area + largura*comprimento;
22         }
23     }
24 }
```

UNIDADE 4

1.

```
01 #include <stdio.h>
02 #define max 5
03 int main()
04 {
05     int i, j, iguais, vet_A[max], vet_B[max];
06     for (i = 0; i<max; i++) // inicializacao
07     {
08         vet_A [i] = 0;
09         vet_B [i] = 0;
10     }
11     for (i = 0; i < max; i++) // valores do vetor A
12     {
13         printf("Digite o %d valor do vetor A: ",
14             i+1);
15         scanf("%d", &vet_A[i]);
16     }
17     printf ("\n");
18     for (i = 0; i<max; i++) // valores do vetor B
19     {
20         printf("Digite o %d valor do vetor B: ",
21             i+1);
22         scanf("%d", &vet_B[i]);
23     }
24     for (i = 0; i<max; i++) // comparacao
25     {
26         for (j=0; j<max; j++){
27             if (vet_A[i] == vet_B[j]){
28                 iguais++;
29             }
30             if (iguais < 1){
31                 printf("\nElementos diferentes: ");
32                 printf("%d ", vet_A[i]);
33             }
34             iguais = 0;
35         }
36     }
37     return (0);
38 }
```

2.

```
01 #include <stdio.h>
02 int main()
03 {
04     int vetor[3], i, j, aux;
05     for (i = 0; i < 3; i++) // inicializacao
06     {
07         vetor[i] = 0;
08     }
09     aux = 0;
10     for (i = 0; i < 3; i++) // valores do vetor
11     {
12         printf ("Digite o %d valor do vetor: ", i);
13         scanf ("%d", &vetor[i]);
14     }
15     for (i = 0; i < 2; i++) // ordenacao
16     {
17         for (j = i+1; j < 3; j++)
18         {
19             if (vetor[i] < vetor[j])
20             {
21                 aux = vetor[i];
22                 vetor[i] = vetor[j];
23                 vetor[j] = aux;
24             }
25         }
26     }
27     for (i = 0; i < 3; i++) // valores do vetor
28     {
29         printf ("vetor[%d] = %d\n", i, vetor[i]);
30     }
31     return (0);
32 }
```

3.

```
01 #include <stdio.h>
02 #include <string.h>
03 #define max 10
04 int main()
05 {
06     char palavra[max];
07     int i;
08     char letra;
09     printf("Digite uma palavra: ");
10     scanf("%s", palavra);
11     strlwr(palavra);
12     if (strlen(palavra) %2 == 1){
13         for (i = 0; i < max; i++){
14             letra = palavra[i];
15             switch (letra) {
16                 case 'a' : printf ("%c ", letra);
17                     break;
18                 case 'e' : printf ("%c ", letra);
19                     break;
20                 case 'i' : printf ("%c ", letra);
21                     break;
22                 case 'o' : printf ("%c ", letra);
23                     break;
24                 case 'u' : printf ("%c ", letra);
25                     break;
26             }
27         }
28     }
29 }
```

4.

```
01 #include <stdio.h>
02 int main()
03 {
04     char palavra[10];
05     int repeticoes, i;
06     printf ("Digite uma palavra: ");
07     scanf ("%s", &palavra);
08     printf ("Digite o número de repetições: ");
09     scanf ("%d", &repeticoes);
10     for (i = 1; i <= repeticoes; i++)
11     {
12         printf("%s\n", palavra);
13     }
14     printf("\n");
15     return (0);
16 }
```

5.

```
01 #include <stdio.h>
02 int main()
03 {
04     int matriz_A[5][5], matriz_B[5][5], i, j, k;
05     printf ("\nMatriz A\n");
06     for (i = 0; i < 5; i++) // matriz A
07     {
08         for (j = 0; j < 5; j++)
09         {
10             printf ("[%d, %d]: ", i+1, j+1);
11             scanf ("%d", &matriz_A[i][j]);
12         }
13     }
14     printf ("\nMatriz B\n");
15     for (i = 0; i < 5; i++) // matriz B
16     {
17         for (j = 0; j < 5; j++)
18         {
19             printf ("[%d, %d]: ", i+1, j+1);
```

```

20         scanf ("%d", &matrix_B[i][j]);
21     }
22 }
23 printf ("\nSoma de Matrizes\n");
24 for (i = 0; i < 5; i++) // Soma
25 {
26     printf ("|\t");
27     for (j = 0; j < 5; j++)
28     {
29         printf ("%d\t",
30             matriz_A[i][j]+matriz_B[i][j]);
31     }
32     printf (" |\n");
33 }
34 printf ("\nSubtração de Matrizes\n");
35 for (i = 0; i < 5; i++) // Subtração
36 {
37     printf ("|\t");
38     for (j = 0; j < 5; j++)
39     {
40         printf ("%d\t",
41             matriz_A[i][j]-matriz_B[i][j]);
42     }
43     printf (" |\n");
44 }

```

6.

```

01 #include <stdio.h>
02 #include <string.h>
03 typedef struct dados {
04     char nome[20];
05     float nota[4];
06     float media;
07 }cadastro;
08 int main() {
09     cadastro boletim[5];
10     cadastro boletimTmp;

```

```
11     int i, j;
12     float soma, notas = 0;
13     for (i=0; i<5; i++){
14         j = 0;
15         printf("\n\nDigite o %dº nome: ", i+1);
16         scanf("%20[^\\n]s", boletim[i].nome);
17         fflush(stdin);
18         for (j=0; j<4; j++){
19             printf("Digite a %dª nota: ", j+1);
20             scanf("%f", &boletim[i].nota[j]);
21             fflush(stdin);
22             soma = soma + boletim[i].nota[j];
23         }
24         boletim[i].media = soma/4;
25         soma = 0;
26         printf("\n** %s - %.0lf - %.0lf - %.0lf - "
27                "%.0lf\n", boletim[i].nome,
28                boletim[i].nota[0], boletim[i].nota[1],
29                boletim[i].nota[2], boletim[i].nota[3]);
30     }
31     for (i=0; i<4; i++){
32         for (j=i+1; j<5; j++){
33             if (boletim[i].media > boletim[j].media){
34                 boletimTmp = boletim[i];
35                 boletim[i] = boletim[j];
36                 boletim[j] = boletimTmp;
37             }
38         }
39     }
40     for (i=0; i<5; i++){
41         printf ("\n%s - %.0lf\n", boletim[i].nome,
42                boletim[i].media);
43         printf ("%.0lf; %.0lf; %.0lf; %.0lf",
44                boletim[i].nota[0], boletim[i].nota[1],
45                boletim[i].nota[2], boletim[i].nota[3]);
46     }
47 }
```

7.

```
01 #include <stdio.h>
02 #define max 20
03 #define N 40
04 typedef struct dados {
05     char titulo[N];
06     char autor[N];
07     char editora[N];
08     int edicao;
09     int ano;
10 } livro;
11 int main()
12 {
13     livro livros[max];
14     int i;
15     for (i = 0; i < max; i++)
16     {
17         printf ("\nDigite os dados do %d livro\n",
18                 i+1);
19         printf ("Titulo: ");
20         scanf ("%40[^\\n]s", livros[i].titulo);
21         fflush (stdin);
22         printf ("Autor: ");
23         scanf ("%40[^\\n]s", livros[i].autor);
24         fflush (stdin);
25         printf ("Editora: ");
26         scanf ("%40[^\\n]s", livros[i].editora);
27         fflush (stdin);
28         printf ("Edição: ");
29         scanf ("%d", &livros[i].edicao);
30         fflush (stdin);
31         printf ("Ano: ");
32         scanf ("%d", &livros[i].ano);
33         fflush (stdin);
34         printf ("\n");
35     }
36     printf ("\n\n** Livros Cadastrados **\n");
37     for (i = 0; i < max; i++)
38     {
39         printf ("Titulo: %s\n", livros[i].titulo);
40         printf ("Autor: %s\n", livros[i].autor);
```

```
40         printf ("Editora: %s\n", livros[i].editora);
41         printf ("Edição: %d\n", livros[i].edicao);
42         printf ("Ano: %d\n", livros[i].ano);
43         printf ("\n");
44     }
45     printf ("\n");
46     return (0);
47 }
```

8.

```
01 #include <stdio.h>
02 #define max 2
03 #define N 30
04 typedef struct dados {
05     char nome[N];
06     char telefone[N];
07     char email[N];
08 } cadastro;
09 int main()
10 {
11     cadastro cadastrados[max];
12     int i;
13     for (i = 0; i < max; i++)
14     {
15         printf("\nDigite os dados do %d cadastro\n",
16                i+1);
17         printf("Nome: ");
18         scanf("%30[^\\n]s", cadastrados[i].nome);
19         fflush(stdin);
20         printf("Telefone: ");
21         scanf("%30[^\\n]s", cadastrados[i].telefone);
22         fflush(stdin);
23         printf("E-mail: ");
24         scanf("%30[^\\n]s", cadastrados[i].email);
25         fflush(stdin);
26     }
27     printf ("\n\n** Pessoas Cadastradas **\n");
28     for (i = 0; i < max; i++)
29     {
30         printf ("Nome: %s\n", cadastrados[i].nome);
```

```
30         printf ("Telefone: %s\n",
31             cadastros[i].telefone);
32         printf ("E-mail: %s\n", cadastros[i].email);
33         printf ("\n");
34     }
35     printf ("\n");
36 }
```

UNIDADE 5

1.

```
01 #include <stdio.h>
02
03 int numero;
04
05 void verifica()
06 {
07     if (numero > 0)
08         printf ("Positivo\n");
09     else if (numero < 0)
10         printf ("Negativo\n");
11     else
12         printf ("Zero\n");
13 }
14
15 int main()
16 {
17     printf("Digite um número: \n");
18     scanf("%d", &numero);
19     verifica();
20     printf ("\n");
21     system ("pause");
22     return (0);
23 }
```

2.

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int soma(int num1, int num2)
05 {
06     int resultado, i;
07     for (i = num1; i <= num2; i++)
08     {
09         resultado = resultado + i;
10     }
11     return (resultado);
12 }
13
14 int main()
15 {
16     int numero1, numero2;
17
18     printf("Digite o primeiro número: ");
19     scanf("%d", &numero1);
20     printf("Digite o segundo número: ");
21     scanf("%d", &numero2);
22     numero1 = soma(numero1, numero2);
23     printf("A soma dos números dados é: %d\n",
24            numero1);
25     system("pause");
26 }
```

3.

```
01 #include <stdio.h>
02
03 int verifica(char letra);
04
05 int main()
06 {
07     char letra;
08     int resultado;
09     printf ("Digite uma letra: ");
10     letra = getch();
11     resultado = verifica(letra);
12     if (resultado == 1)
13         printf ("A letra '%c' é uma vogal\n", letra);
14     else if (resultado == 0)
15         printf ("A letra '%c' é uma consoante, número ou
16             caractere especial\n", letra);
17     system ("pause");
18     return (0);
19 }
20
21 int verifica(char letra)
22 {
23     switch (letra) {
24         case 'a' : return (0);
25             break;
26         case 'e' : return (0);
27             break;
28         case 'i' : return (0);
29             break;
30         case 'o' : return (0);
31             break;
32         case 'u' : return (0);
33             break;
34         case 'A' : return (0);
35             break;
36         case 'E' : return (0);
37             break;
```

```
37         case 'I' : return (0);
38             break;
39         case 'O' : return (0);
40             break;
41         case 'U' : return (0);
42             break;
43         default : return (1);
44             break;
45     }
46 }
```

4.

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int somatorio(int valor);
05
06 int main()
07 {
08     int numero, resultado;
09     printf("Digite um número: ");
10     scanf("%d", &numero);
11     resultado = somatorio(numero);
12     printf("O somatório dos %d primeiros números
13 pares é: %d\n", numero, resultado);
14     system("pause");
15     return(0);
16 }
17
18 int somatorio(int valor)
19 {
20     int soma, i;
21     soma = 0;
22     for (i = 0; i <= valor; i=i+2)
23     {
24         soma = soma + i;
25     }
26     return (soma);
27 }
```

5.

```
01 #include <stdio.h>
02
03 #define max 20
04
05 int verifica(char palavra[]);
06
07 int main(){
08     char palavra[max];
09     int i, vogais;
10     for (i=0; i<max; i++)
11         palavra[i] = '\0';
12     printf("Digite uma palavra: ");
13     fflush(stdin);
14     scanf("%s", palavra);
15     vogais = verifica(palavra);
16     printf("Total de vogais na palavra: %d\n",
17            vogais);
18     return 0;
19 }
20
21 int verifica(char palavra[max])
22 {
23     int i, vogais = 0;
24     char letra;
25     for (i = 0; i < max; i++){
26         letra = palavra[i];
27         switch (letra) {
28             case 'a' : vogais++;
29                         break;
30             case 'e' : vogais++;
31                         break;
32             case 'i' : vogais++;
33                         break;
34             case 'o' : vogais++;
35                         break;
36             case 'u' : vogais++;
37                         break;
```

```
37         case 'A' : vogais++;
38             break;
39         case 'E' : vogais++;
40             break;
41         case 'I' : vogais++;
42             break;
43         case 'O' : vogais++;
44             break;
45         case 'U' : vogais++;
46             break;
47         default : break;
48     }
49 }
50 return vogais;
51 }
```

6.

```
01 #include <stdio.h>
02
03 float converte(float reais, float cotacao);
04
05 int main()
06 {
07     float resultado, valor, cotacao;
08     resultado = 0; // inicializacao
09     printf("Digite o valor da cotação de um dólar em
reais:");
10     scanf("%f", &cotacao);
11     printf("Digite o valor em reais para conversão:
");
12     scanf("%f", &valor);
13     resultado = converte(valor, cotacao);
14     printf("R$%.2f em dólares: %.2f\n", valor,
resultado);
15     return (0);
16 }
17
18 float converte(float reais, float cotacao)
19 {
20     float conversao;
```

```
21     conversao = reais / cotacao;  
22     return (conversao);  
23 }
```

7.

```
01 #include <stdio.h>  
02  
03 #define max 10  
04  
05 int main()  
06 {  
07     char palavra[max];  
08     int i, j;  
09     FILE *arq;  
10     char *arquivo = "texto.txt";  
11     arq = fopen ("texto.txt", "w");  
12     if (arq == NULL)  
13         printf ("Arquivo não aberto. Erro\n");  
14     else  
15     {  
16         for (i=0; i<max; i++)  
17             printf("Digite a %dª palavra: ", i+1);  
18             scanf("%s", palavra);  
19             fflush(stdin);  
20             fprintf(arq, "%s\n", palavra);  
21     }  
22 }  
23 fclose(arq);  
24  
25 arq = fopen ("texto.txt", "r");  
26 if (arq == NULL)  
27     printf ("\nArquivo não aberto. Erro\n");  
28 else  
29 {  
30     while((fgets(palavra, sizeof(palavra),  
31     arq))!=NULL )  
32         printf("%s", palavra);  
33     }  
34     fclose(arq);  
35     return (0);  
36 }
```



anotações



anotações