Professora Me. Gislaine Camila Lapasini Leal

ALGORITMOS E LÓGICA DE PROGRAMAÇÃO II

GRADUAÇÃO

ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

SISTEMAS PARA INTERNET

MARINGÁ-PR

2012

Reitor: Wilson de Matos Silva
Vice-Reitor: Wilson de Matos Silva Filho
Pró-Reitor de Administração: Wilson de Matos Silva Filho
Presidente da Mantenedora: Cláudio Ferdinandi

NEAD - Núcleo de Educação a Distância

Diretoria do NEAD: Willian Victor Kendrick de Matos Silva
Coordenação Pedagógica: Gislene Miotto Catolino Raymundo
Coordenação de Polos: Diego Figueiredo Dias
Coordenação Comercial: Helder Machado
Coordenação de Tecnologia: Fabrício Ricardo Lazilha
Coordenação de Curso: Danilo Xavier Saes
Supervisora do Núcleo de Produção de Materiais: Nalva Aparecida da Rosa Moura
Capa e Editoração: Daniel Fuverki Hey, Fernando Henrique Mendes, Luiz Fernando Rokubuiti e Thayla Daiany Guimarães Cripaldi
Supervisão de Materiais: Nádila de Almeida Toledo
Revisão Textual e Normas: Cristiane de Oliveira Alves, Janaína Bicudo Kikuchi, Jaquelina Kutsunugi e Maria Fernanda Canova Vasconcelos

Ficha catalográfica elaborada pela Biblioteca Central - CESUMAR

C397

CENTRO UNIVERSITÁRIO DE MARINGÁ. Núcleo de Educação a distância:
Algoritmos e lógica de programação II / Gislaine Camila Lapa sini Leal. Maringá - PR, 2012.
199 p.

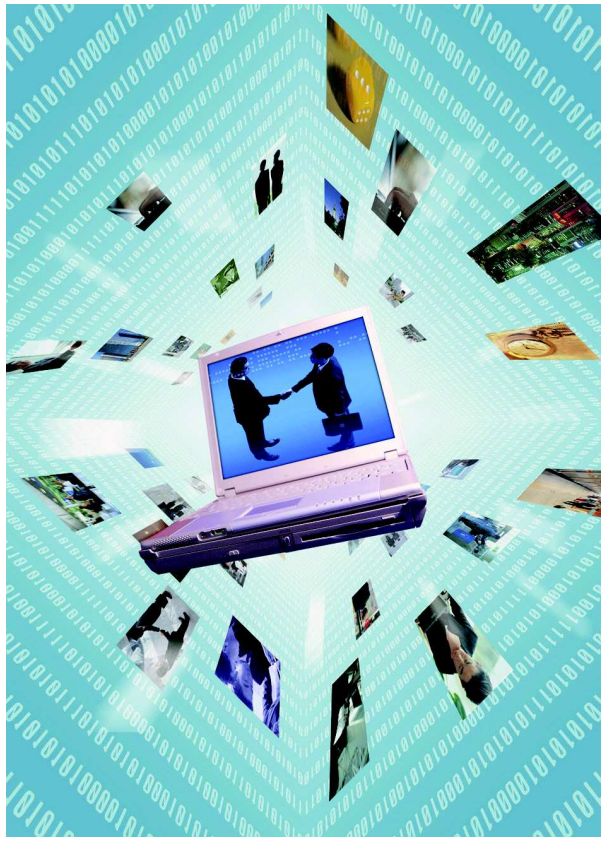
"Graduação em Análise e Desenvolvimento de Sistemas e Sistemas para Internet- EaD".

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

CDD - 22 ed. 005.1
CIP - NBR 12899 - AACR/2

ALGORITMOS E LÓGICA DE PROGRAMAÇÃO II

Professora Me. Gislaine Camila Lapasini Leal



APRESENTAÇÃO DO REITOR

Viver e trabalhar em uma sociedade global é um grande desafio para todos os cidadãos. A busca por tecnologia, informação, conhecimento de qualidade, novas habilidades para liderança e solução de problemas com eficiência tornou-se uma questão de sobrevivência no mundo do trabalho.

Cada um de nós tem uma grande responsabilidade: as escolhas que fizermos por nós e pelos nossos fará grande diferença no futuro.

Com essa visão, o Cesumar – Centro Universitário de Maringá – assume o compromisso de democratizar o conhecimento por meio de alta tecnologia e contribuir para o futuro dos brasileiros.

No cumprimento de sua missão – "promover a educação de qualidade nas diferentes áreas do conhecimento, formando profissionais cidadãos que contribuam para o desenvolvimento de uma sociedade justa e solidária" –, o Cesumar busca a integração do ensino-pesquisa -extensão com as demandas institucionais e sociais; a realização de uma prática acadêmica que contribua para o desenvolvimento da consciência social e política e, por fim, a democratização do conhecimento acadêmico com a articulação e a integração com a sociedade.

Diante disso, o Cesumar almeja ser reconhecido como uma instituição universitária de referência regional e nacional pela qualidade e compromisso do corpo docente; aquisição de competências institucionais para o desenvolvimento de linhas de pesquisa; consolidação da extensão universitária; qualidade da oferta dos ensinos presencial e a distância; bem -estar e satisfação da comunidade interna; qualidade da gestão acadêmica e administrativa; compromisso social de inclusão; processos de cooperação e parceria com o mundo do trabalho, como também pelo compromisso e relacionamento permanente com os egressos, incentivando a educação continuada.

Professor Wilson de Matos Silva

Reitor

Caro aluno, "*ensinar não é transferir conhecimento, mas criar as possibilidades para a sua produção ou a sua construção*" (FREIRE, 1996, p. 25). Tenho a certeza de que no Núcleo de Educação a Distância do Cesumar, você terá à sua disposição todas as condições para se fazer um competente profissional e, assim, colaborar efetivamente para o desenvolvimento da realidade social em que está inserido.

Todas as atividades de estudo presentes neste material foram desenvolvidas para atender o seu processo de formação e contemplam as diretrizes curriculares dos cursos de graduação, determinadas pelo Ministério da Educação (MEC). Desta forma, buscando atender essas necessidades, dispomos de uma equipe de profissionais multidisciplinares para que, independente da distância geográfica que você esteja, possamos interagir e, assim, fazer-se presentes no seu processo de ensino-aprendizagem-conhecimento.

Neste sentido, por meio de um modelo pedagógico interativo, possibilitamos que, efetivamente, você construa e amplie a sua rede de conhecimentos. Essa interatividade será vivenciada especialmente no ambiente virtual de aprendizagem – AVA – no qual disponibilizamos, além do material produzido em linguagem dialógica, aulas sobre os conteúdos abordados, atividades de estudo, enfim, um mundo de linguagens diferenciadas e ricas de possibilidades efetivas para a sua aprendizagem. Assim sendo, todas as atividades de ensino, disponibilizadas para o seu processo de formação, têm por intuito possibilitar o desenvolvimento de novas competências necessárias para que você se aproprie do conhecimento de forma colaborativa.

Portanto, recomendo que durante a realização de seu curso, você procure interagir com os textos, fazer anotações, responder às atividades de autoestudo, participar ativamente dos fóruns, ver as indicações de leitura e realizar novas pesquisas sobre os assuntos tratados, pois tais atividades lhe possibilitarão organizar o seu processo educativo e, assim, superar os desafios na construção de conhecimentos. Para finalizar essa mensagem de boas-vindas, lhe estendo o convite para que caminhe conosco na Comunidade do Conhecimento e vivencie a oportunidade de constituir-se sujeito do seu processo de aprendizagem e membro de uma comunidade mais universal e igualitária.

Um grande abraço e ótimos momentos de construção de aprendizagem!

Professora Gislene Miotto Catolino Raymundo

Coordenadora Pedagógica do NEAD- CESUMAR

APRESENTAÇÃO

Livro: ALGORITMOS E LÓGICA DE PROGRAMAÇÃO II

Professora Me. Gislaine Camila Lapasini Leal

Caro(a) aluno(a)! Bem-vindo à disciplina de Algoritmos e Lógica de Programação II. Sou a professora Gislaine Camila e nesta disciplina aprenderemos 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. É uma linguagem que vem se popularizando por ser uma linguagem de propósito geral e não ser vinculada a um hardware específico ou qualquer outro sistema.

Apresento 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 verifique o funcionamento dos mesmos. Apenas a leitura dos exemplos não é suficiente, o aprendizado requer prática.

Na Unidade I veremos o histórico da linguagem C, suas características, 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, constantes, realizar operações de atribuição, entrada e saída de dados. Conheceremos, também, as palavras reservadas da linguagem C, os operadores e funções intrínsecas. A partir destes conteúdos iniciaremos a construção de nossos primeiros programas em C.

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

A Unidade III aborda a construção de programas com repetição de um 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 sobre as diferenças de cada uma delas e como utilizá-las.

Na Unidade IV serão apresentados os conceitos sobre estruturas de dados homogêneas (vetores e matrizes) e estruturas de dados heterogêneas (*structs*). Essas estruturas permitem agrupar diversas informações em uma única variável. Aprenderemos como declará-las e manipulá-las nas operações de entrada, atribuição e saída. Trataremos questões relacionadas à pesquisa de um elemento em um vetor e a 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 V, trabalharemos com a modularização de nossos programas utilizando funções. Abordaremos os conceitos relacionados ao escopo de variáveis e a passagem de parâmetros por valor e por referência, protótipo de função e recursividade. Estudaremos as funções disponíveis para manipulação de arquivos que nos permitem abrir um arquivo, verificar erro durante a abertura, verificar fim de arquivo, fechar um arquivo, 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. Preste atenção nos momentos **REFLITA**, pois eles apresentam informações importantes que você precisa pensar com calma e que te auxilia a consolidar o conhecimento adquirido. Além disso, serão apresentadas **ATIVIDADES DE AUTOESTUDO** que permitem que você coloque em prática os conhecimento e **EXERCÍCIOS DE FIXAÇÃO**, que apresentam exercícios resolvidos que podem te auxiliar no aprendizado e sanar eventuais dúvidas.

Desejo a você um bom estudo!

SUMÁRIO

UNIDADE III

UNIDADE IV

UNIDADE V

FUNÇÕES E ARQUIVOS

UNIDADE I

CONCEITOS INICIAIS

Professora Me. Gislaine Camila Lapasini Leal

Objetivos de Aprendizagem

- Conhecer os conceitos iniciais de programação, da linguagem C e a estrutura de um programa em C.
- Estudar os tipos de dados, variáveis, constantes, expressões e operadores.
- Entender como realizar atribuição, entrada e saída de dados.

Plano de Estudo

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

- **Linguagem C**
- **Conceitos Iniciais de Programação**
- **Estrutura de um Programa em C**
- **Identificadores**
- **Tipos de Dados**
- **Palavras Reservadas**
- **Variáveis**
- **Constantes**
- **Expressões e Operadores**
- **Funções Intrínsecas**

- **Atribuição**
- **Entrada de Dados**
- **Saída de Dados**
- **Construindo um programa**

INTRODUÇÃO

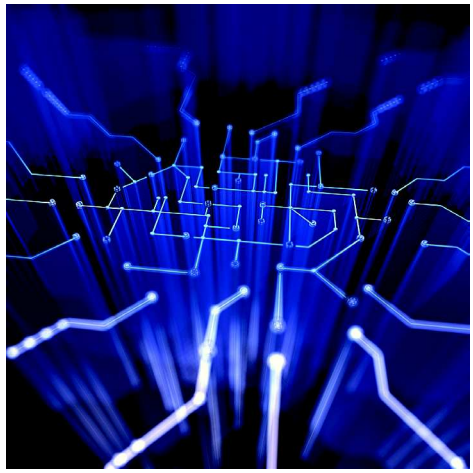
Nesta primeira unidade, você será introduzido ao universo da Linguagem de Programação C, uma linguagem poderosa e que tem se tornado cada vez mais popular.

Primeiramente, você conhecerá o histórico da linguagem C, suas características, pontos fortes e fracos e compiladores disponíveis para essa linguagem. 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 do compilador que adotaremos na construção de nossos programas.

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

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

LINGUAGEM C



Fonte: SHUTTERSTOCK.COM

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 (KERNINGHAN; 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 qualquer outro sistema. De modo que é fácil escrever programas que serão executados sem mudanças em qualquer máquina que suporta C (KERNINGHAN; RITCHIE, 1988).

Segundo Rocha (2006), a principal característica da linguagem C é que ela combina as vantagens de uma linguagem de alto nível com a eficiência da linguagem de montagem *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 com que programadores experientes escrevam códigos mais compactos

Anotações

e eficientes, possibilita que programadores inexperientes realizam construções sem sentido, as quais são aceitas como válidas pelo compilador. Deste modo, ao construir programas utilizando C devemos ficar atento 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.

Na literatura podemos encontrar diversos compiladores C, sendo os principais: GCC, Dev C++, C++ Builder, Turbo C e Visual C#. Em nossa disciplina de Algoritmos e Lógica de Programação II adotaremos o Turbo C (Figura 1).

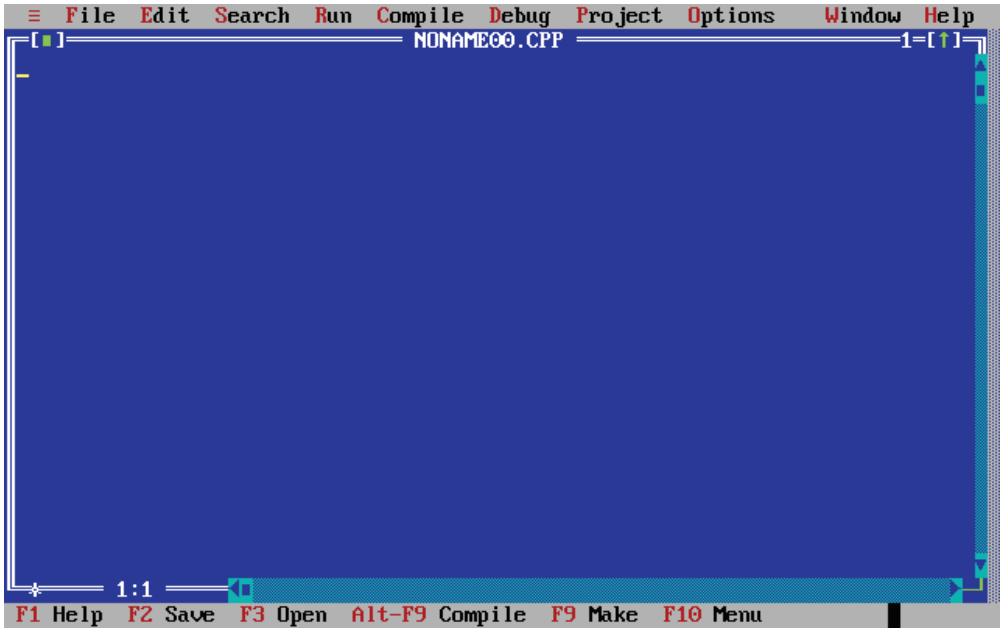


Figura 1- Interface Turbo C

Anotações

CONCEITOS INICIAIS DE PROGRAMAÇÃO

A programação inicia-se com a escrita do programa (código-fonte) e encerra com a geração de um programa executável. A escrita do programa deve ser realizada em um editor de textos. Após a criação do programa temos que compilá-lo. O processo de compilação analisa o código 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 lincador (ligador) reúne o programa objeto com as bibliotecas referenciadas e gera o código executável (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.

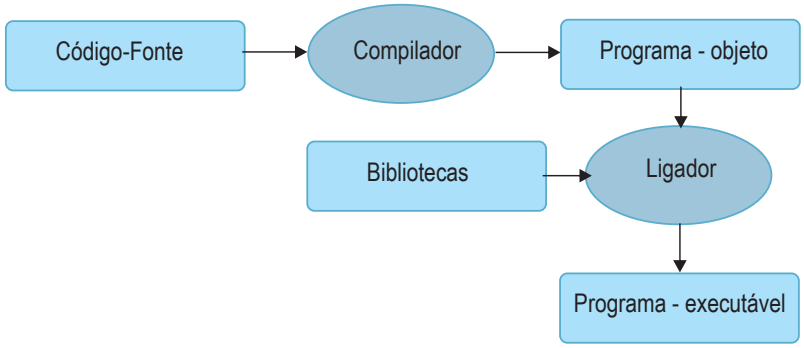


Figura 2 - Etapas para a criação de um programa

Fonte: adaptado de Rocha (2006)

Em nossa disciplina escreveremos nossos programas utilizando o Turbo C (Figura 1). Na Figura 3 são mostrados os menus em que as operações compilar, ligar e executar são realizadas.

Anotações

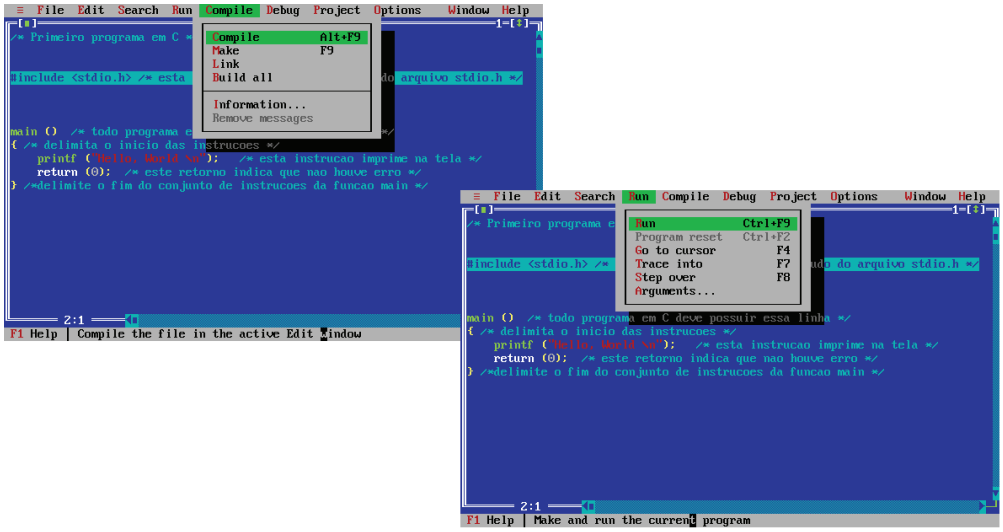


Figura 3 - Turbo C

ESTRUTURA DE UM PROGRAMA EM C

Kerninghan e Ritchie (1988) destacam que a única maneira de aprender uma nova linguagem de programação é escrevendo programas nela. Com isto, para entender a estrutura de um programa em C, vamos construir nosso primeiro programa "*Hello, world*" (Quadro 1).

Quadro 1 - Programa HelloWorld

```
# include  <stdio.h>
main()
{
    printf("Hello, World");
    return (0);
}
```

Anotações

Fique tranquilo!! Vamos analisar o programa acima linha a linha e entender o que cada um destes elementos significa.

Na primeira linha temos a instrução **#include** <stdhio.h>, que indica ao compilador que deve incluir o conteúdo do arquivo pré-definido chamado de stdio.h. Este arquivo é chamado de 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 uma função denominada **main**. Os programas em C são formados por chamadas de função. Obrigatoriamente, todo programa deve possuir uma função **main**, a qual é chamada quando o programa é executado. Note que o conteúdo da função é delimitado por chaves, de modo análogo ao início e fim do algoritmo. 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 dela é **printf,** que é uma função previamente definida no arquivo de cabeçalho stdio.h que imprime na tela a sequência de caracteres “Hello, World”. A última linha, **return** (0), indica o valor de retorno da função. No caso 0, indica que o programa terminou sem erros. Observe que ao final de cada instrução há um ponto e vírgula, isto é, todos os 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 Turbo C, conforme Figura 4. Observe que além do código apresentado no Quadro 1, há outras informações que estão entre /* */. Este comando é utilizado para escrever comentários em nosso código, isto é, o compilador desconsidera qualquer coisa que esteja entre estes dois pares de símbolos. 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

Anotações

entender o funcionamento dos programas.

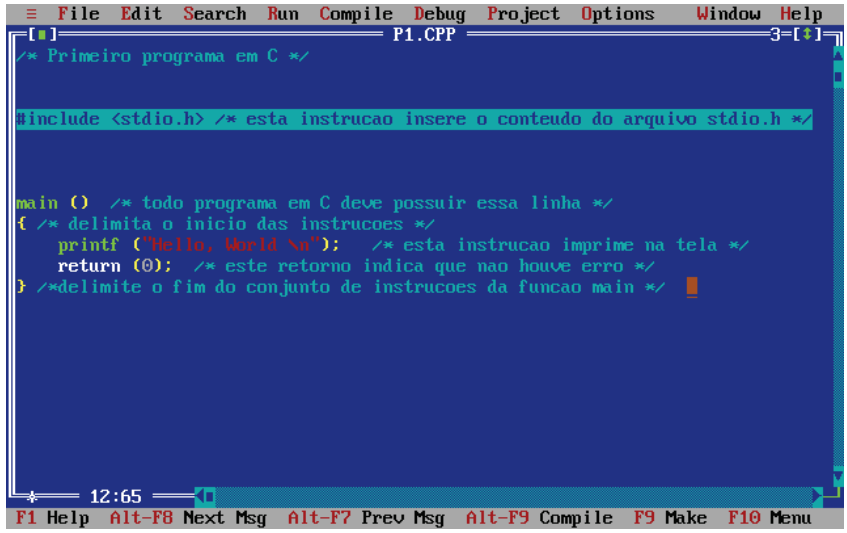


Figura 4 - Primeiro programa em C – Código

Após escrever o código do programa temos que compilá-lo (ALT + F9). É neste momento que serão realizadas as verificações de sintaxe. E para visualizar o resultado do programa basta executá-lo. No caso do Turbo C podemos ir ao menu *Run* ou pressionar as teclas Ctrl + F9.

Ao executar o código temos como resultado a impressão na tela da cadeia de caracteres Hello, World, conforme pode ser visualizado na Figura 5.

Anotações

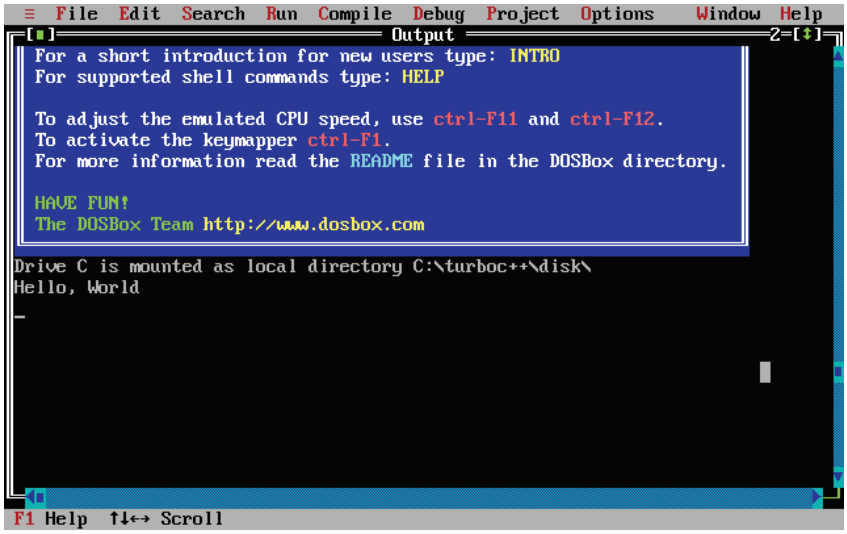


Figura 5 - Primeiro programa em C – Saída

A estrutura geral de um programa em C é apresentada no Quadro 2.

Quadro 2 - Estrutura geral de um programa em C

```
main()
{
   conjunto de instruções;
    return (0);
}
```

Anotações

IDENTIFICADORES

Fonte: SHUTTERSTOCK.COM

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, que começa com uma letra ou sublinha. Em geral, evita-se iniciar um identificador com sublinhas, pois este tipo de notação 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. Os comandos em C só podem ser escritos em minúsculos, senão o compilador irá interpretá-los como variáveis. O Quadro 3 apresenta alguns exemplos de identificadores válidos e inválidos.

Anotações

Quadro 3 - Exemplo de identificadores

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

Devemos lembrar que em C o identificador “A” não é o mesmo que o identificador “a”, cada um é um identificador único.

TIPOS DE DADOS

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

O tipo **char** é composto por caracteres simples e *strings* (cadeia de caracters). 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. O tipo **pointer** representa um tipo especial, que não contém informação, mas sim, uma localização de memória que 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 o tamanho

Anotações

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 os tipos como sem sinal, duplicando assim a gama de valores que pode ser representado. O modificador **short** reduz a capacidade de armazenamento, enquanto o modificador **long** aumenta a capacidade.

Quadro 4 - Tipos de dados e faixa de valores

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

Fonte: Ascencio; 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 acima estão em conformidade com o padrão ANSI.

Anotações

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.

No Quadro 5 são destacadas as palavras reservadas da linguagem C. Lembre-se: não podemos utilizar como identificador uma palavra reservada.

Quadro 5 - Palavras reservadas da linguagem C

Palavras reservadas			
asm	template	do	register
catch	thhis	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	

Fonte: Pappas; Murray (1991)

Anotações

VARIÁVEIS

Em nossos programas precisamos armazenar algumas informações e para isto 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 contendo vários elementos do tipo **char** (ASCENCIO; CAMPOS, 2010).

A sintaxe para declaração de variáveis é dada por:

<tipo> identificador;

No Quadro 6 são apresentados exemplos de declaração de variáveis. Atente para o fato de que a declaração de variáveis é seguida de ; . Observe, também, que podemos declarar em uma mesma linha diversas variáveis do mesmo tipo.

Quadro 6 - Exemplos de declaração de variáveis

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

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 (inteiro e real) podem ser utilizadas em expressão desse tipo.

O Quadro 7 apresenta os operadores aritméticos da linguagem C, destacando suas representações e forma de uso.

Quadro 7 - Operadores aritméticos

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	%	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.

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).

Quadro 8 - Operadores matemáticos de atribuição

Operador	Exemplo	Explicação
+=	x += y	Equivale a x = x + y.
-=	x -=y	Equivale a x = x - y.
***=**	x *=y	Equivale a x = x * y.
/=	x /=y	Equivale a x = x / y.
%=	x %=y	Equivale a x = x % y.
++	x++	Equivale a x = x + 1.
	y = ++x	Equivale a x = x + 1 e depois y = x.
	y = ++ x	Equivale a y = x e depois x = x + 1.
--	x--	Equivale a x = x - 1.
	y = -- x	Equivale a x = x - 1 e depois y = x.
	y = x --	Equivale a y = x e depois x = x - 1.

Fonte: adaptado de (ASCENCIO; CAMPOS, 2010)

As expressões **relacionais** referem-se à comparação entre dois valores de um tipo básico. Os operadores relacionais são destacados no Quadro 9, em que é possível visualizar o operador, símbolo associado e forma de uso.

Anotações

Quadro 9 - Operadores relacionais

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

Fonte: adaptado de (ASCENCIO; CAMPOS, 2010)

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 expressões matemáticas.

No Quadro 10 são descritos cada um dos operadores lógicos: conjunção, disjunção e negação.

Anotações

Quadro 10 - Operadores lógicos

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

Fonte: adaptado de (LOPES; GARCIA, 2002)

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

Anotações

Quadro 11 - Níveis de Precedência de Operador

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

Fonte: adaptado de (PAPPAS; MURRAY, 1991)

FUNÇÕES INTRÍNSECAS


Fonte: SHUTTERSTOCK.COM

As funções intrínsecas são fórmulas matemáticas prontas que podemos utilizar em nossos programas. O Quadro 12 apresenta as principais funções da linguagem C, destacando o comando associado, um exemplo e o que ela faz.

Quadro 12 - Funções intrínsecas da linguagem C

Função	Exemplo	Objetivo
ceil	ceil(x)	Arredonda um número real para cima. Por exemplo, cel(2.3) é 3.
cos	cos(x)	Calcula o cosseno de x. O valor de x deve estar em radianos.
exp	exp(x)	Obtém o logaritmo natural e elevado 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 base 10 de x.
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.
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 de π

Fonte: Ascencio; Campos (2010)

ATRIBUIÇÃO

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

identificador = expressão

O identificador representa a variável a qual será atribuído um valor e o termo expressão é o que será atribuído, podendo ser uma expressão aritmética ou lógica. Alguns exemplos do uso do comando de atribuição podem ser visualizados no Quadro 13.

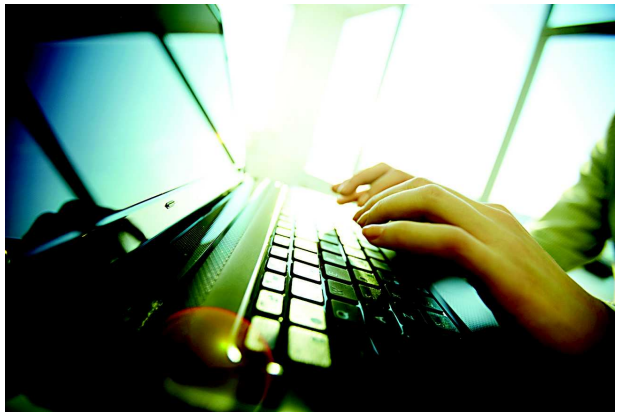
Quadro 13 - Exemplos de atribuição

Exemplo	
x = 3;	O valor 3 é atribuído a variável x.
x = x + 5;	É atribuído a variável x o valor de x mais 5.
x = 2.5;	O valor 2.5 é atribuído a variável x.
sexo = 'M' ,	O valor 'M' é atribuído a variável denominada sexo.
nome = "Maria";	O valor "Maria" é atribuído à cadeia de caracteres chamada nome.

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

Anotações

ENTRADA DE DADOS


Fonte: SHUTTERSTOCK.COM

A entrada de dados permite receber os dados digitados pelo usuário. Os dados recebidos são armazenados em variáveis. Na linguagem C existem diversas funções para entrada de dados, algumas delas são **cin**, **gets** e **scanf** (Quadro 14) (ASCENCIO; CAMPOS, 2010).

Os comandos **gets** e **scanf** armazenam toda a cadeia de caracteres até que seja pressionada a tecla ENTER. Já o comando **cin** armazena os caracteres até que seja encontrado um espaço em branco.

Quadro 14 - Exemplos de entrada de dados

Exemplo	
cin >> x;	Um valor digitado pelo usuário é armazenado na variável x.
gets(nome);	Um ou mais caracteres digitados pelo usuário são armazenados na variável nome.
scanf(&x);	Um valor digitado pelo usuário é armazenado na variável x.

Fonte: adaptado de (ASCENCIO; CAMPOS, 2010)

Anotações

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

scanf ("expressão de controle", lista de argumentos);

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 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 identificar do vetor já é o endereço do primeiro elemento do vetor.

Quadro 15 - Especificadores de formato

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 *string*.
%x	Leitura de um número em base hexadecimal.
%p	Leitura de um ponteiro.

Fonte: adaptado de (ROCHA, 2006)

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

scanf (“%d”, &idade);

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 dados aos usuários. Na linguagem C 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. A lista de argumentos pode conter identificadores de variáveis, expressões aritméticas ou lógicas e valores constantes.

Além dos especificadores de formato podemos utilizar códigos especiais na saída de dados. Esses códigos são apresentados no Quadro 16.

Quadro 16 - Códigos especiais

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

Fonte: adaptado de (PAPPAS; MURRAY, 1991)

O programa (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.

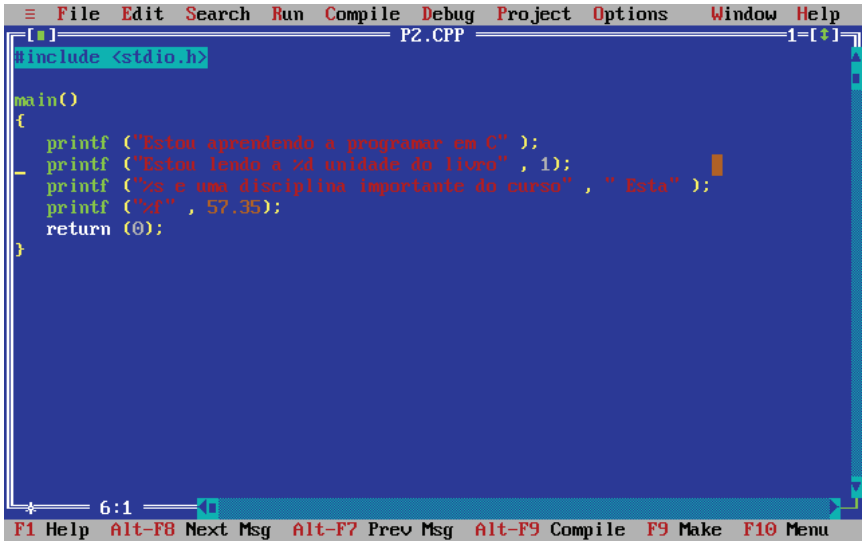


Figura 6 - Entendendo a função printf – Código

Anotações

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.

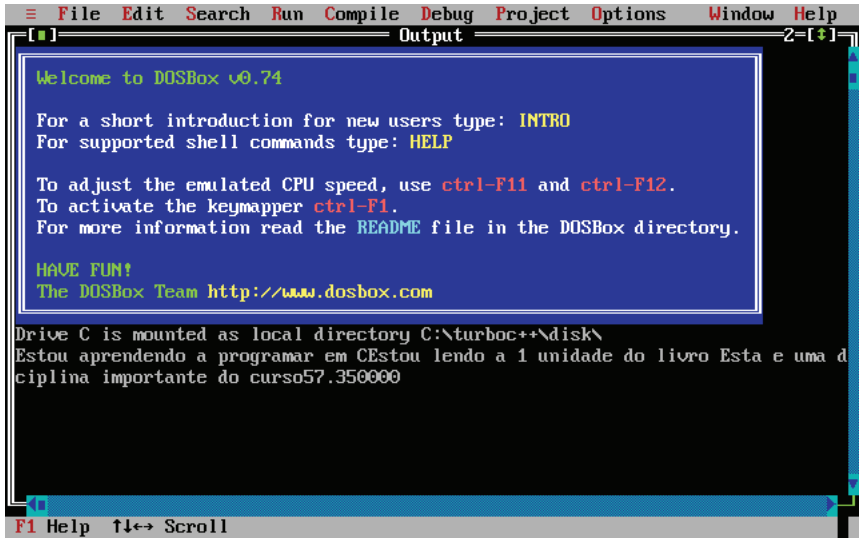


Figura 7 - Entendendo a função printf – Saída

Ao comparar os códigos da Figura 6 e Figura 8 podemos observar que foi inserido o código especial \n na função **printf**. Esse código é responsável por posicionar o cursor em uma nova linha. Com isto, temos que a saída é a impressão de cada mensagem em uma linha, como pode ser visualizado na Figura 9.

Anotações

Figura 8 - Entendendo a função printf - Código

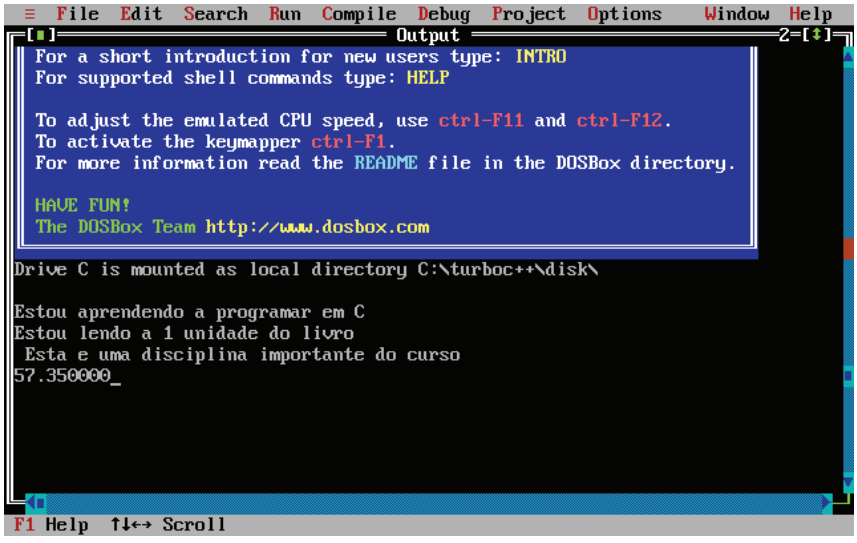


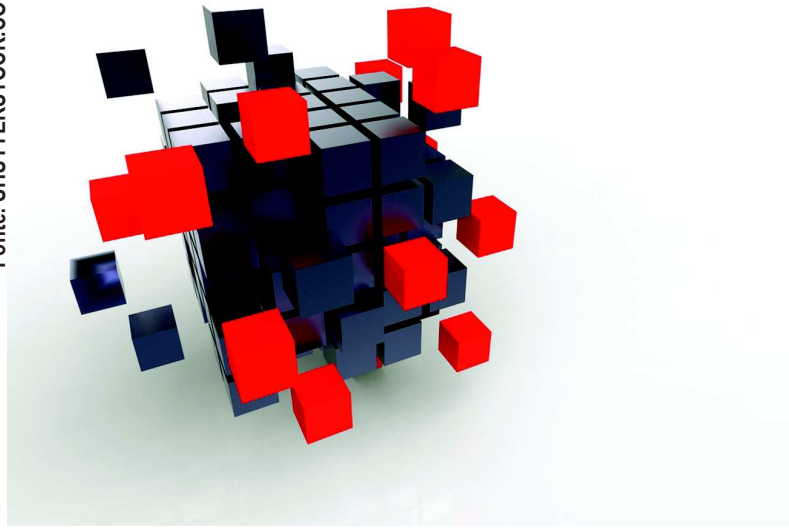
Figura 9 - Entendendo a função printf - Saída

Anotações

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

CONSTRUINDO UM PROGRAMA

Fonte: SHUTTERSTOCK.COM



Vamos elaborar um programa que leia nome, idade e altura de uma pessoa e exiba nome, idade, altura e ano de nascimento da pessoa. Para facilitar o entendimento sobre o problema vamos estruturá-lo em três partes: Entrada – Processamento e Saída.

Na entrada de dados temos que obter os dados de nome, idade e altura. Cada um desses dados precisa ser armazenado em uma variável e na leitura deles 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 o nome, idade, altura e ano de nascimento. Para mostrar essas informações no vídeo utilizaremos a função **printf**.

Anotações

Agora que entendemos o problema podemos partir para a construção do programa. Você se recorda da estrutura básica de um programa em C? Nosso programa deve ter uma função **main** e as instruções têm que estar ente os colchetes.

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, deste modo precisamos de uma variável do tipo **char.** A variável altura é do tipo **float** e idade do tipo **int**.

O Quadro 17 apresenta o código para o programa acima descrito.

Anotações

Quadro 17 - Programa em C

```
#include <stdio.h>  /* insere o conteudo do arquivo stdio.h */
main()
{  /* declaracao das variaveis */
   int    idade, ano;
   float  altura;
   char   nome[30];
   /*entrada de dados */
   printf (“Digite o seu nome: ”); /*mensagem ao usuario */
   scanf (“%s”, nome); /* leitura do nome */
   printf (“Digite  a idade:”); /*mensagem ao usuario */
   scanf (“%d”, &idade);  /* leitura da idade */
   printf (“Digite a altura: ”);/*mensagem ao usuario */
   scanf (“%f”, &altura); /* leitura da altura*/
   /* processamento */
   ano = 2012 - idade;  /*calculo do ano de nascimento */
   /*saida de dados */
   printf (“ \nO nome e : %s”, nome);
   printf (“ \nA altura e : %f”, altura);
   printf (“ \nA idade e : %d”, idade);
   printf (“\nO ano de nascimento e : %d ”, ano);
   return (0);
}
```

Anotações

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).

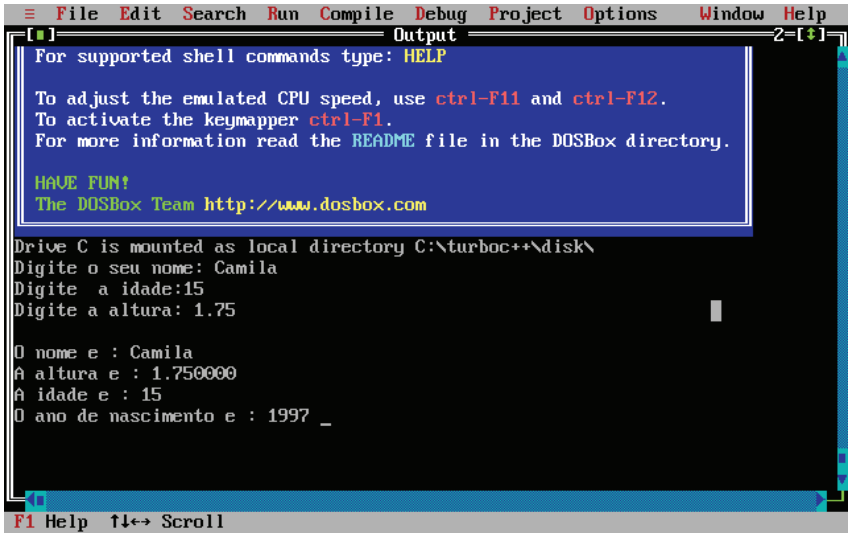


Figura 10 - Programa em C - Saída

No entanto, podemos formatar 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 2 casas decimais. Na função **printf** ao utilizar o especificador de formato inserimos um ponto e o número de casas decimais desejado.

Anotações

Quadro 18 - Programa em C

```
#include <stdio.h>  /* insere o conteudo do arquivo stdio.h */
main()
{  /* declaracao das variaveis */
   int    idade, ano;
   float  altura;
   char   nome[30];
   /*entrada de dados */
   printf (“Digite o seu nome: ”); /*mensagem ao usuario */
   scanf (“%s”, nome); /* leitura do nome */
   printf (“Digite  a idade:”); /*mensagem ao usuario */
   scanf (“%d”, &idade);  /* leitura da idade */
   printf (“Digite a altura: ”);/*mensagem ao usuario */
   scanf (“%f”, &altura); /* leitura da altura*/
   /* processamento */
   ano = 2012 - idade;  /*calculo do ano de nascimento */
   /*saida de dados */
   printf (“ \nO nome e : %s”, nome);
   printf (“ \nA altura e : %.2f”, altura);
   printf (“ \nA idade e : %d”, idade);
   printf (“\nO ano de nascimento e : %d ”, ano);
   return (0);
}
```

Anotações

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

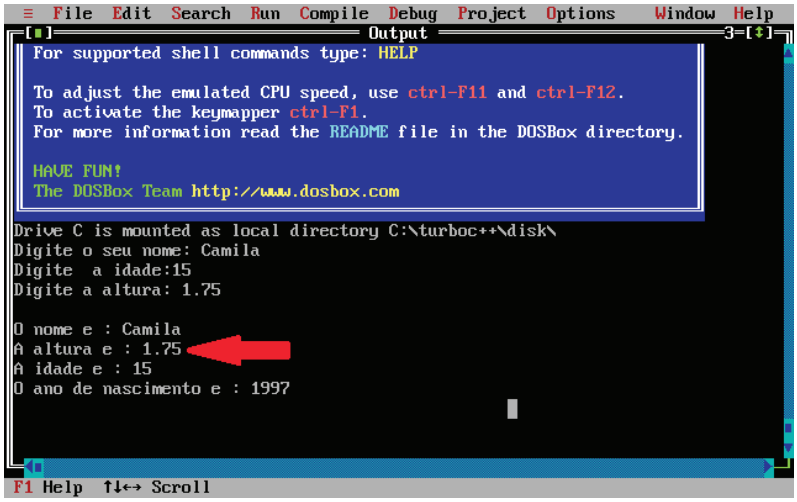


Figura 11 - Programa em C - Saída Formatada

CONSIDERAÇÕES FINAIS

Nesta unidade, você deu os primeiros passos no aprendizado sobre a Linguagem de Programação C, conhecendo as características, 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 chamado 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 das instruções devemos inserir a função **return** (0), que o programa terminou sem erros e que todos os comandos terminam com ponto e vírgula.

Aprendemos como documentar nossos códigos inserindo comentários os quais são escritos entre /* */ e podem ter mais de uma linha. Entendemos as regra para a nomeação de identificadores.

Estudamos os sete tipos básicos de dados (**char**, **int**, **float**, **double**, **enum**, **void** e **pointer**), operadores e funções intrínsecas disponíveis na linguagem C. Em relação aos tipos de dados vimos que eles podem gerar outros tipos a partir da aplicação dos modificadores **unsigned**, **short** e **long**.

Entendemos como realizar a atribuição, entrada de dados e saída de dados. O comando usado para atribuição é representado por = . A entrada de dados 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.

Saiba mais sobre o **Assunto**

Para conhecer um pouco mais sobre o histórico da linguagem C, acesse o texto disponível em: **<http://pt.wikipedia.org/wiki/C_%28linguagem_de_programa%C3%A7%C3%A3o%29>.**

Para entender como instalar o Turbo C no Windows Vista e Seven, acesse o vídeo disponível em: <http://www.youtube.com/watch?v=3FXAwCv6amU>.

Reflita

A linguagem C é case sensitive.

ATIVIDADE DE AUTOESTUDO

1. Marque com um X os identificadores incorretos.

( ) idade	( ) media idade	( ) 2nome
( ) nome*r	( ) x2	( ) 012
( ) media_peso%	( ) endereco+cep	( ) /fone
( ) aluno_nota	( ) A	( ) 1234P

2. Escreva um programa que leia o nome de uma pessoa e escreva a seguinte mensagem "Bem-vindo à disciplina de Algoritmos e Lógica de Programação II, Fulano".

3. Escreva um programa que lê um número positivo inteiro e apresente o quadrado e a raiz quadrada deste número.

4. Escreva um programa que recebe quatro números inteiros, calcula e apresenta a média aritmética entre eles. Observação: não esqueça de formatar o valor da média no momento de apresentá-lo, utilize 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= $2\pi r$. Dica: utilize as funções intrínsecas vistas na unidade.

6. Identifique os erros no programa abaixo.

Quadro 19 - Programa em C

```
#include <stdio.h>  /* insere o conteudo do arquivo stdio.h
main()
{  /* declaracao das variaveis */
   int    num1, num2, total
   /*entrada de dados */
   printf (“Digite o primeiro numero: ”); /*mensagem ao usuario */
   scanf (“%d”, num1); /* leitura do primeiro numero */
   printf (“Digite  o segundo numero:”); /*mensagem ao usuario */
   scanf (“%d”, &num2);  /* leitura do segundo numero*/
   /* processamento */
   total = num1 + num2   /*calculo do ano de nascimento */
   /*saida de dados */
   printf (“\n A soma dos números e : %d ”, soma);
   return (0);
}
```

EXERCÍCIOS DE FIXAÇÃO

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

Quadro 20 - Programa em C

```
#include <stdio.h>
main ()
{
   int   num, ant, suc;
   printf(“ Digite o numero:”);
   scanf (“%d”, &num);
   ant = num - 1;
   suc = num +1;
   printf(“\n O antecessor e: %d”, ant);
   printf(“\n O sucessor e: %d”, suc);
   return (0);
}
```

Quadro 21 - Programa em C

```
#include <stdio.h>
main ()
{
   int   num;
   printf(“ Digite o numero:”);
   scanf (“%d”, &num);
   printf(“\n O antecessor e: %d”, num -1);
   printf(“\n O sucessor e: %d”, num+1);
   return (0);
}
```

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

Quadro 22 - Programa em C

```
#include <stdio.h>
main ()
{
   float   n1, n2, n3, n4, media;
   printf(“ Digite a nota 1:”);
   scanf (“%f”, &n1);
   printf(“ Digite a nota 2:”);
   scanf (“%f”, &n2);
   printf(“ Digite a nota 3:”);
   scanf (“%f”, &n3);
   printf(“ Digite a nota 4:”);
   scanf (“%f”, &n4);
   media = (n1 + n2+ n3+ n4)/4;
   printf(“ A media e: %.2f”, media);
   return (0);
}
```

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).

Quadro 23 - Programa em C

```
#include <stdio.h>
main ()
{
   float   deposito, taxa, rendimento, total;
   printf(“ Informe o valor do deposito:”);
   scanf (“%f”, &deposito);
   printf(“\n Informe a taxa de juros:”);
   scanf (“%f”, &taxa);
   rendimento = deposito * (taxa/100);
   total = deposito + rendimento;
   printf(“\n O rendimento e: %.2f”, rendimento);
   printf(“\n O total e: %.2f”, total);
   return (0);
}
```

4. Escreva um programa que receba dois números, calcule e apresente um elevado ao outro.

Quadro 24 - Programa em C

```
#include <stdio.h>
#include <math.h>
main ()
{
   float   num1, num2, total;
   printf(“ Informe o primeiro numero:”);
   scanf (“%f”, &num1);
   printf(“\n Informe o segundo numero:”);
   scanf (“%f”, &num2);
   total = pow(num1, num2);
   printf(“\n %.2f elevado a %.2f e: %.2f”, num1, num2, total);
   return (0);
}
```

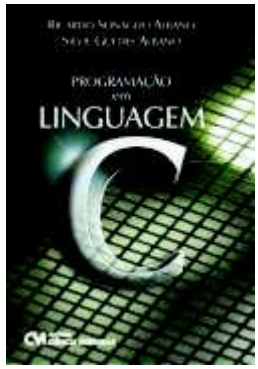
5. Elabore um programa que calcule a área de um trapézio.

Quadro 25 - Programa em C

```
#include <stdio.h>
main ()
{
   float   base1, base2, altura, area;
   printf(“ Informe o valor da base maior:”);
   scanf (“%f”, &base1);
   printf(“\n Informe o valor da base menor:”);
   scanf (“%f”, &base2);
   printf(“\n Informe o valor da altura:”);
   scanf (“%f”, &altura);
   area = ((base1 + base2) * altura)/2;
   printf(“\n A area do trapezio e: %.2f”, area);
   return (0);
}
```

Leitura Complementar

ALBANO, R. S. **Programação em Linguagem C**. Editora Ciência Moderna, 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, técnicos ou cursos livres. Além disso, os autodidatas poderão utilizar este livro, já que o mesmo 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 vai lendo o livro. O mesmo 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. Salientando que todos os exercícios possuem resolução contida no final do livro.

UNIDADE II

ESTRUTURA CONDICIONAL

Professora Me. Gislaine Camila Lapasini Leal

Objetivos de Aprendizagem

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

Plano de Estudo

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

- **Estrutura condicional**
- **Estrutura condicional simples**
- **Estrutura condicional composta**
- **Estrutura case**

INTRODUÇÃO

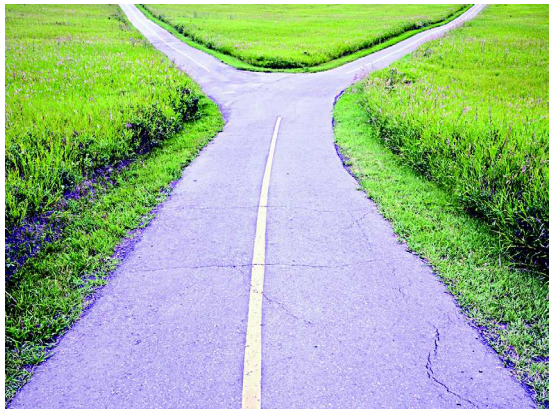
Nesta unidade, você estudará a estrutura condicional conhecendo a estrutura condicional simples, composta e estrutura case. Com os conhecimentos adquiridos na unidade I conseguíamos construir programas sequenciais, isto é, que a partir da entrada, os dados eram processados e apresentávamos algumas informações na saída.

Na estrutura condicional podemos impor condições para a execução de uma instrução ou conjunto de instruções, isto é, podemos criar condições que permitem desviar o fluxo de execução de um programa. Para construir essas condições utilizaremos os conceitos de variáveis e expressões vistos na unidade I. Estudaremos a estrutura condicional simples que nos permite tomar uma decisão. A estrutura condicional composta que a partir de uma expressão, podemos seguir dois caminhos, um quando o resultado do teste é verdadeiro e outro quando o resultado é falso. E a estrutura case que é uma generalização da estrutura condicional composta em que pode haver uma ou mais condições a serem testadas e cada uma delas pode ter uma instrução diferente associada.

Ao estudar cada estrutura condicional veremos um exemplo e construiremos algoritmos para visualizar a aplicação dos conceitos estudados.

Ao final desta unidade estaremos aptos a construir programas com desvio de fluxo, aumentando assim, o leque de problemas que podemos solucionar.

ESTRUTURA CONDICIONAL


Fonte: SHUTTERSTOCK.COM

A estrutura condicional é fundamental para qualquer linguagem de programação, uma vez que sem elas o fluxo seria seguido sequencialmente, sem nenhum desvio, ou seja, instrução a instrução. A estrutura condicional possibilita o desvio do fluxo do programa, sendo também denominada de estrutura de seleção ou estrutura de controle (MANZANO e OLIVEIRA, 1997; ASCENCIO e 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 seções seguintes estudaremos as estruturas condicionais da Linguagem C.

ESTRUTURA CONDICIONAL SIMPLES

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

Anotações

A sintaxe do comando é:

```
if (<condição>)
{
<instruções para 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, o algoritmo segue o primeiro comando após "}". 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. Não podemos esquecer de inserir o ";" ao final de cada instrução.

Agora que conhecemos a sintaxe da estrutura condicional simples vamos construir 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 26 apresenta o programa em C para o problema descrito acima. Lembre-se que os textos compreendidos entre /* */ correspondem a comentários e não executados.

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 variáveis 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**)

consiste em verificar se o valor do resto da divisão do número lido por dois é igual a zero. Se este valor for verdadeiro é executado o conjunto de instruções delimitado por { }, 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 for falso é executada a linha com o comando return (0).

Quadro 26 - Programa em C

```
/* Estrutura condicional simples em C */
#include <stdio.h> /* esta instrucao insere o conteudo do arquivo stdio.h */
#include <math.h> /*insere o conteudo do arquivo math, que tem a funcao
sqrt, calcula da raiz quadrada */
main ()  /* todo programa em C deve possuir essa linha */
{ /* delimita o inicio das instrucoes */
   int num;
   float raiz;
   printf(“ Digite um numero inteiro:”);
   scanf (“%d”, &num);
   if (num % 2 == 0 )
   {
     raiz = sqrt(num);
     printf (“A raiz quadrada e: %.3f”, raiz);
   }
   return (0);  /* este retorno indica que nao houve erro */
} /*delimite o fim do conjunto de instrucoes da funcao main */
```

Vamos analisar o resultado da simulação do programa para os valores 18 (Figura 12) e 15 (Figura 13), respectivamente. Como 18 é um número par o resultado do teste lógico é

Anotações

verdadeiro, com isto temos a impressão do valor da raiz quadrada, no caso 4.243. Observe que o valor apresentado possui três casas decimais devido à formatação definida no comando **printf** (%.3f).

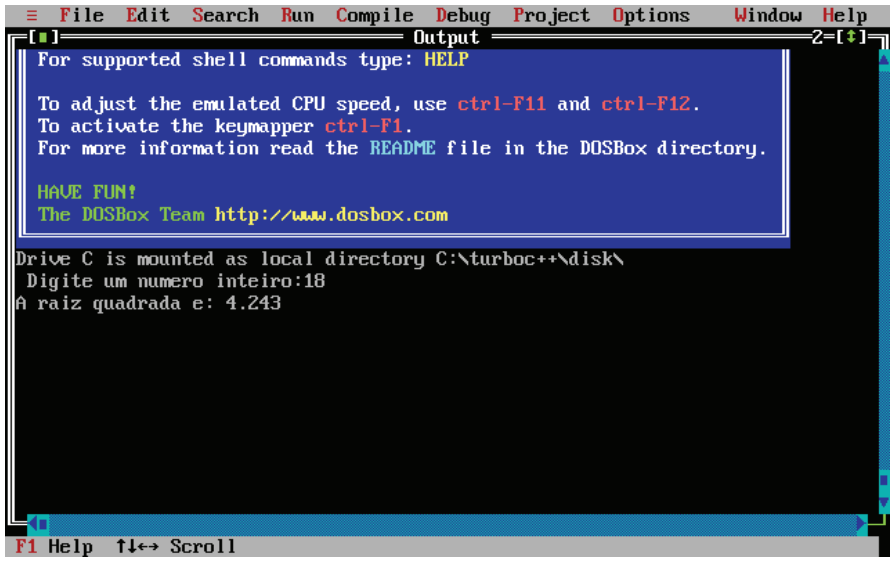


Figura 12 - Programa em C - Saída

No caso do valor 15, o resultado do teste lógico é falso. Deste modo, as instruções compreendidas entre { } não são executadas, portanto temos a execução do return (0).

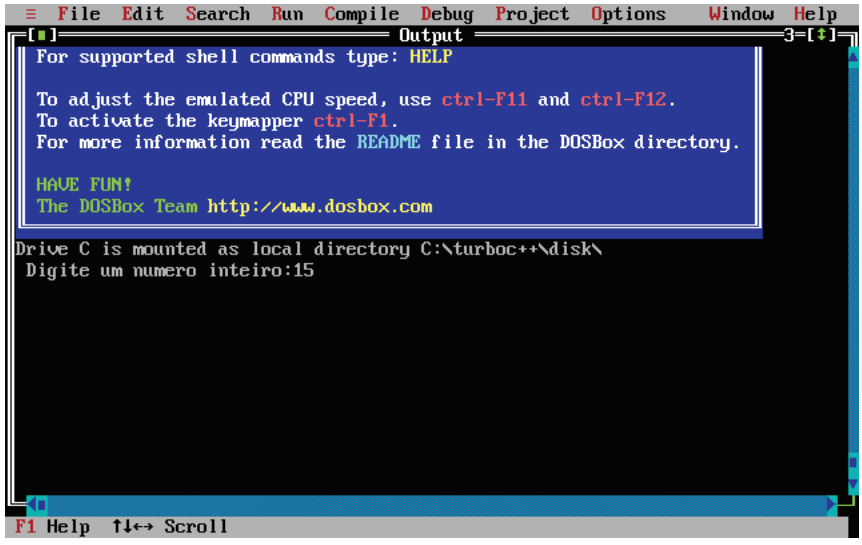


Figura 13 - Programa em C - Saída

ESTRUTURA CONDICIONAL COMPOSTA

Na estrutura condicional composta é realizada a avaliação de uma única expressão lógica. Se o resultado dessa avaliação for verdadeiro é executado a instrução ou o conjunto de instruções compreendido entre o comando entre as chaves após o **if**. Se o resultado da avaliação for falso é executado 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 vamos construir 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 27 é apresentado um programa em C que verifica se o número é par ou ímpar. Em relação às variáveis foram declaradas três variáveis num, quadrado e raiz. O número inteiro obtido do usuário é armazenado na variável num.

O teste lógico realizado na estrutura condicional verifica se o valor do resto da divisão do número lido por dois é igual a zero. Se este valor for verdadeiro é executado o conjunto de instruções após o **if** delimitado por { }, isto é, temos a execução da instrução de atribuição do

Anotações

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 for falso é executado o conjunto de instruções após o **else** delimitado por { } em que é calculado o valor de num ao quadrado.

Quadro 27 - Programa em C

```
/* Estrutura condicional composta em C */
#include <stdio.h> /* esta instrucao insere o conteudo do arquivo stdio.h
*/
#include <math.h> /*insere o conteudo do arquivo math, que tem a funcao
sqrt que efetua o calculo da raiz quadrada */
main ()  /* todo programa em C deve possuir essa linha */
{ /* delimita o inicio das instrucoes */
   int quadrado, num;
   float raiz;
   printf(“ Digite um numero inteiro:”);
   scanf (“%d”, &num);
   if (num % 2 == 0 )
   {
     raiz = sqrt(num);
     printf (“O numero e par”);
     printf (“\nA raiz quadrada e: %.3f”, raiz);
   }
   else
   {
     quadrado = num * num;
     printf (“O numero e impar”);
     printf (“\nO numero ao quadrado e: %d”, quadrado);
   }
   return (0);  /* este retorno indica que nao houve erro */
} /*delimite o fim do conjunto de instrucoes da funcao main */
```

Tomemos como exemplo os valores 15 (Figura 14) e 16 (Figura 15), respectivamente. Vamos analisar o resultado da execução do programa para o valor 15. Na avaliação do teste lógico temos que o resto da divisão de 15 por 2 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 14.

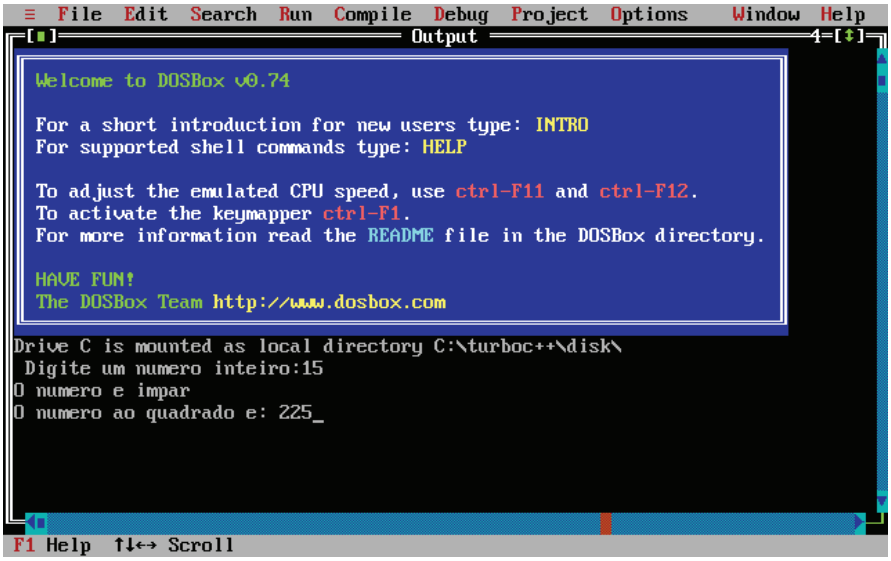


Figura 14 - Programa em C - Saída

Ao analisar 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 15.

Anotações

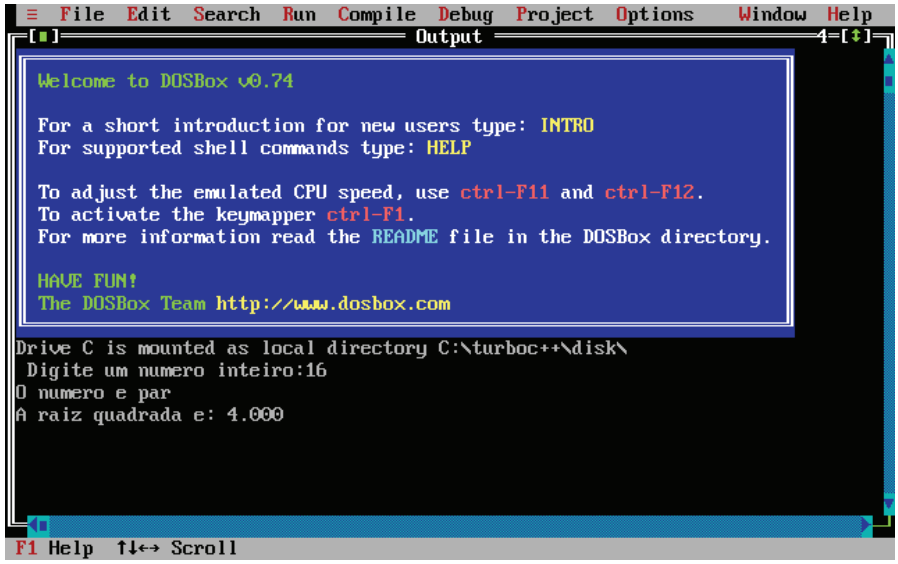


Figura 15 - Programa em C - Saída

ESTRUTURA CASE

Fonte: SHUTTERSTOCK.COM

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, é uma construção de

Anotações

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, ou seja, 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 dessa estrutura é:

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

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

Lopes e Garcia (2002) destacam que a estrutura case é bastante utilizado na construção de menus, tornando-os mais claros.

Anotações

Para ilustrar o funcionamento da estrutura **case** vamos construir um programa que permita ao usuário escolher que operação (1 – soma, 2 – subtração, 3 – multiplicação e 4 – divisão) deseja realizar com dois números. 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 28.

Anotações

Quadro 28 - Programa em C

```
/* Estrutura case em C */
#include <stdio.h> /* esta instrucao insere o conteudo do arquivo stdio.h */
main ()  /* todo programa em C deve possuir essa linha */
{ /* delimita o inicio das instrucoes */
   float num1, num2, resultado;
   int   op;
   printf(" Digite o primeiro numero:");
   scanf ("%f", &num1);
   printf("\n Digite o segundo numero:");
   scanf ("%f", &num2);
   printf("\nEscolha a operacao: \n 1 - Soma \n 2 - Subtracao \n 3 - Multiplicacao
\n 4 - Divisao \n" );
   scanf ("%d", &op);
   switch (op)
   {
    case 1 : resultado = num1 + num2;
          printf (" A soma e : %.3f", resultado);
          break;
    case 2 : resultado = num1 - num2;
          printf (" A subtracao e : %.3f", resultado);
          break;
    case 3 : resultado = num1 * num2;
          printf (" A multiplicacao e : %.3f", resultado);
          break;
    case 4 : resultado = num1 / num2;
          printf (" A divisao e : %.3f", resultado);
          break;
    default : printf("\n Opcao invalida");
   }
   return (0);  /* este retorno indica que nao houve erro */
} /*delimite o fim do conjunto de instrucoes da funcao main */
```

O resultado obtido na execução do programa para cada uma das operações é ilustrado nas Figuras 16, 17, 18 e 19.

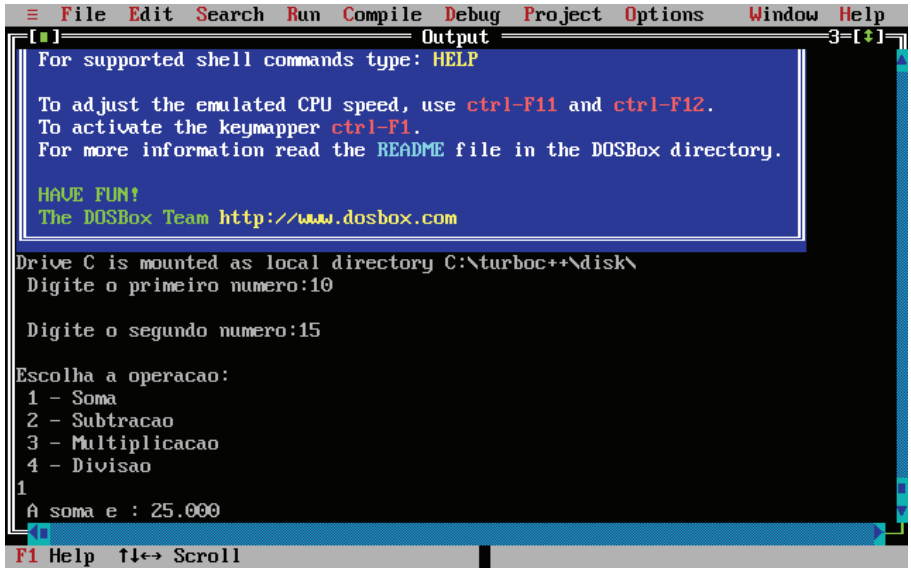


Figura 16 - Programa em C - Saída

Anotações

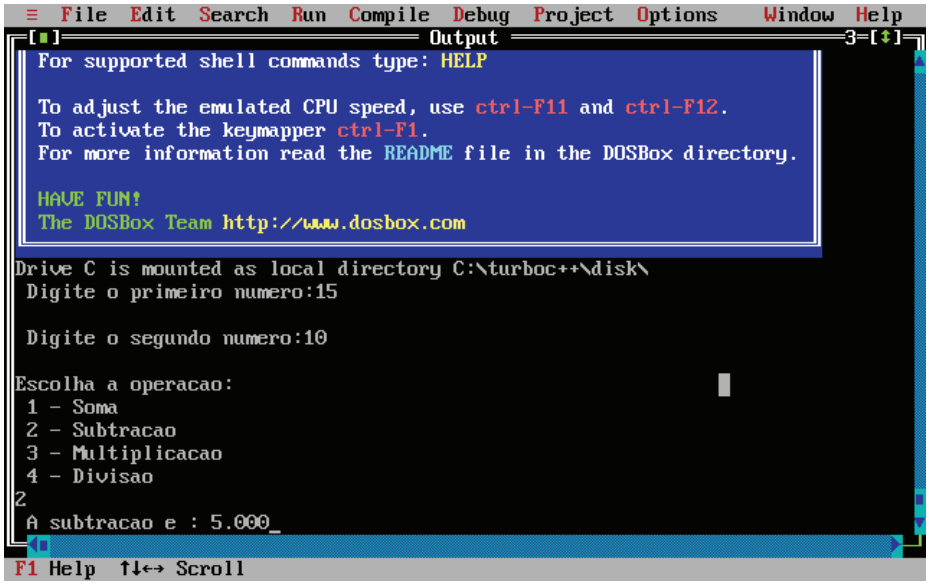


Figura 17 - Programa em C - Saída

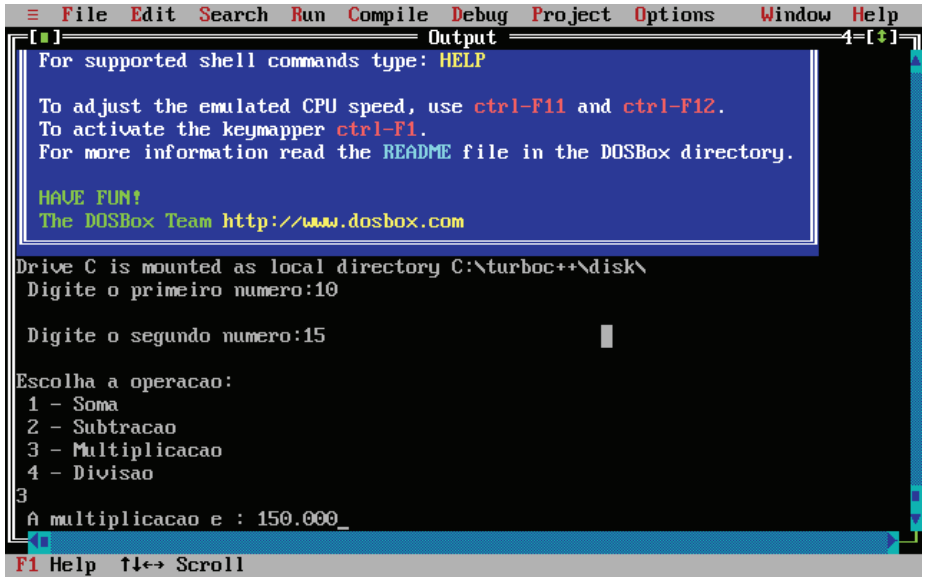


Figura 18 - Programa em C - Saída

Anotações

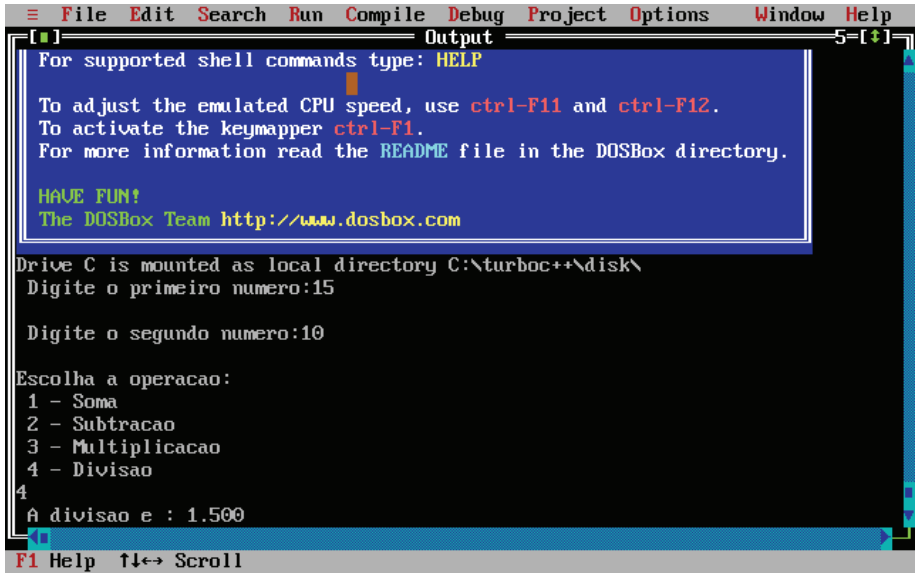


Figura 19 - Programa em C - Saída

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 29.

Anotações

Quadro 29 - Programa em C

```
#include <stdio.h> /* esta instrucao insere o conteudo do arquivo stdio.h */
main ()  /* todo programa em C deve possuir essa linha */
{ /* delimita o inicio das instrucoes */
   float num1, num2, resultado;
   int   op;
   printf(“ Digite o primeiro numero:”);
   scanf (“%f”, &num1);
   printf(“\n Digite o segundo numero:”);
   scanf (“%f”, &num2);
  printf(“\nEscolha a operacao: \n 1 - Soma \n 2 - Subtracao \n 3 - Multiplicacao
\n 4 - Divisao \n” );
   scanf (“%d”, &op);
   if (op == 1) {
      resultado = num1 + num2;
      printf (“ A soma e : %.3f”, resultado);
   }
   else {
     if (op == 2) {
          resultado = num1 - num2;
          printf (“ A subtracao e : %.3f”, resultado);
     }
     else {
          if (op == 3) {
              resultado = num1 * num2;
              printf (“ A multiplicacao e : %.3f”, resultado);
          }
          else
          {
               resultado = num1 / num2;
               printf (“ A divisao e : %.3f”, resultado);

 }
       }
    }
   return (0);  /* este retorno indica que nao houve erro */
} /*delimite o fim do conjunto de instrucoes da funcao main */
```

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 30.

Quadro 30 - Estruturas condicionais

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 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.

O Quadro 31 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 das estruturas.

Neste ponto você deve estar se perguntando qual a diferença entre os programas apresentados no Quadro 28, Quadro 29 e Quadro 31. Os três programas são soluções para o problema descrito, o que os diferencia é a eficiência. A eficiência pode ser analisada em função do número de operações, no caso comparações que o programa deve realizar.

Quadro 31 - Programa em C

```
#include <stdio.h> /* esta instrucao insere o conteudo do arquivo stdio.h */
main ()  /* todo programa em C deve possuir essa linha */
{ /* delimita o inicio das instrucoes */
   float num1, num2, resultado;
   int   op;
   printf(“ Digite o primeiro numero:”);
   scanf (“%f”, &num1);
   printf(“\n Digite o segundo numero:”);
   scanf (“%f”, &num2);
   printf(“\nEscolha a operacao: \n 1 - Soma \n 2 - Subtracao \n 3 -
Multiplicacao \n 4 - Divisao \n” );
   scanf (“%d”, &op);
   if (op == 1) {
     resultado = num1 + num2;
     printf (“ A soma e : %.3f”, resultado);
   }
   if (op == 2) {
     resultado = num1 - num2;
     printf (“ A subtracao e : %.3f”, resultado);
   }
   if (op == 3) {
     resultado = num1 * num2;
     printf (“ A multiplicacao e : %.3f”, resultado);
   }
   if (op ==4) {
     resultado = num1 / num2;
     printf (“ A divisao e : %.3f”, resultado);
   }
   return (0);  /* este retorno indica que nao houve erro */
} /*delimite o fim do conjunto de instrucoes da funcao main */
```

Observe que ao utilizar a estrutura condicional simples (Quadro 31) 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 e o resultado será verdadeiro e as linhas compreendidas entre { } serão executadas. A partir daí os demais testes lógicos também serão avaliados e as instruções não serão executadas por que o resultado será falso. Ao utilizar a estrutura condicional composta ou a estrutura case temos um menor número de testes lógicos para ser avaliado.

Ao analisar o código da estrutura condicional composta (Quadro 29) e estrutura case (Quadro 28) podemos notar que é mais fácil compreender o funcionamento do programa utilizando a estrutura case, pois o código é mais claro. Fique tranquilo!! Com a realização das atividades práticas você conseguirá definir com mais facilidade quando é melhor utilizar cada uma das estruturas.

CONSIDERAÇÕES FINAIS

Nesta unidade, você aprendeu a construir programas em C utilizando a estrutura condicional. Com essa estrutura é possível elaborar programas em que a execução de uma instrução ou um conjunto de instruções está atrelado à avaliação de um teste lógico.

Estudamos a estrutura condicional simples, a estrutura condicional composta 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. A sintaxe dessa estrutura é dada por:

```
if (<condição>)
{
    <instruções para condição verdadeira>;
}
```

Aprendemos que na estrutura condicional composta é realizada a avaliação de uma única expressão lógica, no entanto temos dois caminhos para seguir. Se o resultado desta avaliação

for verdadeiro é executado a instrução ou o conjunto de instruções compreendido entre o comando entre as chaves após o **if**. Se o resultado da avaliação for falso é executado a instrução ou o conjunto de instruções entre chaves após o **else**. A sintaxe desta estrutura é:

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

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 e sua sintaxe é:

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

Vimos que em alguns casos as estruturas condicionais são intercambiáveis, isto é, podemos resolver um mesmo problema utilizando a estrutura condicional simples, a estrutura condicional composta e a estrutura case. Entendemos que a diferença associada ao uso de cada uma delas

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 I 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 você faça as atividades de autoestudo.

Saiba mais sobre o **Assunto**

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

Reflita

A estrutura case só pode ser utilizada em situações mutuamente exclusivas.

ATIVIDADE DE AUTOESTUDO

1. Escreva um programa que receba cinco números inteiros e apresente o maior e o menor.
2. Faça um programa que leia um número e informe se ele é divisível por 3 e por 7.
3. 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 abaixo 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 da carga.

Código estado	Percentual Imposto
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).

EXERCÍCIOS DE FIXAÇÃO

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

Quadro 32 - Programa em C

```
#include <stdio.h>
main ()
{
   int   num, ant, suc;
   printf(“ Digite o numero:”);
   scanf (“%d”, &num);
   ant = num - 1;
   suc = num +1;
   printf(“\n O antecessor e: %d”, ant);
   printf(“\n O sucessor e: %d”, suc);
   return (0);
}
```

2. Faça um programa que leia um número e informe se ele é divisível por 5.

Quadro 33 - Programa em C

```
#include <stdio.h>
main ()
{
   int   num;
   printf(“ Informe o numero:”);
   scanf (“%d”, &num);
   if (num % 5 == 0)
      printf(“\n O numero %d e divisivel por 5”, num);
   else
      printf(“\n O numero %d nao e divisivel por 5”, num);
   return (0);
}
```

3. 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 abaixo 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

Quadro 34 - Programa em C

```
#include <stdio.h>
main ()
{
   char  nome[30];
   int   idade;
   printf(“ Informe o nome:”);
   scanf (“%s”, nome);
   printf(“\n Informe a idade:”);
   scanf (“%d”, &idade);
   if ( idade <= 18) {
       printf (“\n Nome %s :”, nome);
       printf (“\n Idade %d:”, idade);
       printf (“\n O valor do plano e: R$50,00”);
   }
   else
   {
     if ((idade >=19) && (idade <=29))
     {
        printf (“\n Nome %s :”, nome);
        printf (“\n Idade %d:”, idade);
        printf (“\n O valor do plano e: R$70,00”);
     }
     else
     {
      if ((idade >=30) && (idade <=45))
      {
          printf (“\n Nome %s :”, nome);
          printf (“\n Idade %d:”, idade);
          printf (“\n  O valor do plano e: R$90,00”);
      }
      else
      {
           if ((idade >=46) && (idade <=65))
           {
                printf (“\n Nome %s :”, nome);
                printf (“\n Idade %d:”, idade);
                printf (“\n O valor do plano e: R$130,00”);
           }
          else
          {
               printf (“\n Nome %s :”, nome);
               printf (“\n Idade %d:”, idade);
               printf (“\n O valor do plano e: R$170,00”);
          }
      }
     }
   }
   return (0);
}
```

4. 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).

Quadro 35 - Programa em C

```
#include <stdio.h>
main ()
{
   int   idade;
   printf("\n Informe a idade:");
   scanf ("%d", &idade);
   if (idade < 16)
   {
     printf ("Nao eleitor");
   }
   else
   {
        if ((idade < 18) || (idade > 65))
       {
           printf ("Eleitor facultativo");
       }
       else
      {
           printf (" Eleitor obrigatorio");
      }
   }
   return (0);
}
```

5. 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 x altura) -58
Para mulheres	(62.1 x altura) -44.7

Quadro 36 - Programa em C

```
#include <stdio.h>
main ()
{
   float   altura, peso;
   char    sexo;
   printf("\n Informe o sexo (M/F):");
   scanf("%c", &sexo);
   printf("\n Informe a altura:");
   scanf ("%f", &altura);
   if ((sexo== 'F') || (sexo=='f'))
   {
      peso= (62.1 * altura) - 44.7;
   }
   else
   {
      peso =(72.7 *altura)-58;
   }
   printf ("\n O sexo e: %c", sexo);
   printf ("\n A altura e : %.2f", altura);
   printf ("\n O peso ideal e : %.2f", peso);
   return (0);
}
```

6. 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 abaixo.

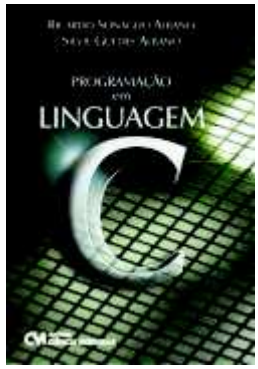
Prato	Bebida
Italiano 750 cal	Chá 30 cal
Japonês 324 cal	Suco de laranja 80 cal
Salvadorenho 545 cal	Refrigerante 90 cal

Quadro 37 - Programa em C

```
#include <stdio.h>
main ()
{
   int    op;
   float  total;
   total = 0;
   printf (“\n 1 - Italiano 2 - Japones 3 - Salvadorenho”);
   printf(“\n Informe o prato desejado:”);
   scanf(“%d”, &op);
   switch (op)
   {
    case 1 : total = total + 750;
             break;
    case 2:  total = total + 324;
             break;
    case 3:  total = total + 545;
             break;
   }
   printf(“\n 1 - Cha 2 - Suco de laranja 3 - Refrigerante:”);
   printf(“\n Informe a bebida desejada:”);
   scanf(“%d”, &op);
   switch (op)
   {
    case 1 : total = total + 30;
            break;
    case 2:  total = total + 80;
            break;
    case 3:  total = total + 90;
            break;
   }
   printf (“\n O total de calorias e : %.2f”, total);
   return (0);
}
```

Leitura Complementar

ALBANO, R. S. **Programação em Linguagem C**. Editora Ciência Moderna, 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, técnicos ou cursos livres. Além disso, os autodidatas poderão utilizar este livro, já que o mesmo 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 vai lendo o livro. O mesmo 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. Salientando que todos os exercícios possuem resolução contida no final do livro.

Anotações

UNIDADE III

ESTRUTURAS DE REPETIÇÃO

Professora Me. Gislaine Camila Lapasini Leal

Objetivos de Aprendizagem

- Estudar as estruturas de repetição controladas e condicionais.
- Construir programas utilizando estruturas de repetição.

Plano de Estudo

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

- **Estrutura de Repetição**
- **Estrutura FOR**
- **Estrutura WHILE**
- **Estrutura DO WHILE**
- **Problema 1**
- **Problema 2**

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 ter que precisar reescrever trechos de códigos idênticos.

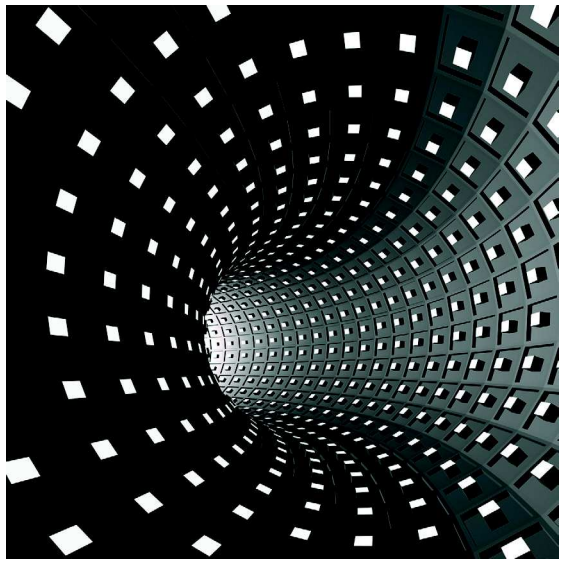
Aprenderemos como utilizar as estruturas de repetição com laços contados e laços condicionais. O uso de laços contados é restrito a situações em que sabemos previamente quantas vezes as instruções precisam ser executadas. Na linguagem C a estrutura de repetição com laço contado é o **for**.

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 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?

Anotações

ESTRUTURA DE REPETIÇÃO



Fonte: SHUTTERSTOCK.COM

Há situações em nossos programas que precisamos repetir um determinado trecho de código ou todo o código um determinado número de vezes. Nestes casos, utilizaremos uma estrutura de repetição que nos permite criar um *loop* para efetuar o processamento de um trecho de código quantas vezes for necessário. Na literatura essas estruturas de repetição (*loop*) são, também, denominadas de 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 número de vezes variável (LOPES; GARCIA, 2002).

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 laços condicionais. O laço contado é utilizado quando conhecemos previamente o número de iterações que precisa ser realizado. Já em um laço condicional este número de iterações

Anotações

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 seções seguintes estudaremos cada uma destas estruturas de repetição da linguagem C, destacando sua sintaxe e aplicação.

ESTRUTURA FOR

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

```
for ( i= valor inicial; condição;
incremento ou decremento de i);
{
    <instruções>;
}
```

Na primeira parte temos a inicialização da variável i, que 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 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 ;.

Anotações

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

No Quadro 38 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 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 esse teste.

Quadro 38 - Programa em C

```
#include <stdio.h>
main ()
{
   char  nome[30];
   int   i ;
   printf("\n Informe o nome:");
   scanf("%s", nome);
   for (i=1; i<=10; i++)
   {
       printf("\n %s", nome);
   }
   return (0);
}
```

O resultado da simulação do programa (Quadro 38) é apresentado na Figura 20.

Anotações

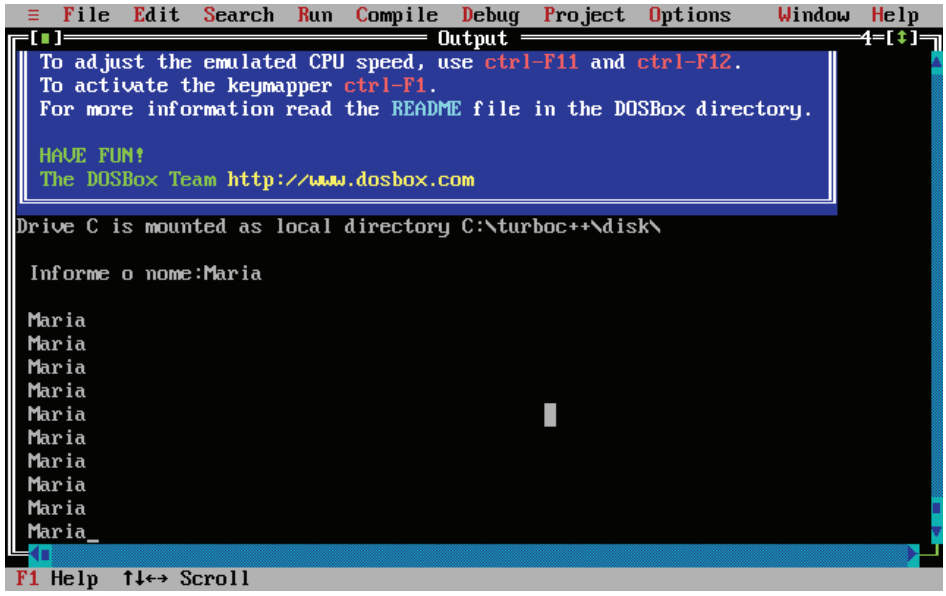


Figura 20 - Programa em C - Saída

Como temos a execução de uma única instrução podemos suprimir as chaves, conforme programa apresentado no Quadro 39.

Quadro 39 - Programa em C

```
#include <stdio.h>
main ()
{
   char  nome[30];
   int   i ;
   printf("\n Informe o nome:");
   scanf("%s", nome);
   for (i=1; i<=10; i++)
       printf("\n %s", nome);
   return (0);
}
```

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

ESTRUTURA WHILE

A estrutura **while** é uma estrutura do tipo laço condicional, isto é, o *loop* baseia-se na análise de uma condição. Essa estrutura é utilizada quando temos um número indefinido de repetições e se caracteriza por realizar um teste condicional no início. Como o teste condicional é executado no início, 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. Vamos analisar o funcionamento desta estrutura utilizando o exemplo apresentado na seção anterior, receber um nome e imprimi-lo dez vezes.

Na estrutura **while** temos que representar o critério de parada (dez iterações) utilizando uma condição. Para tanto, definimos uma variável que irá controlar o número de repetições. Essa variável precisa ser inicializada fora da estrutura de repetição e incrementada no interior do laço. No Quadro 40 é apresentado o programa.

Anotações

Quadro 40 - Programa em C

```
#include <stdio.h>
main ()
{
   char  nome[30];
   int   i ;
   printf("\n Informe o nome:");
   scanf("%s", nome);
   i = 0;
   while (i != 10)
   {
      printf("\n %d - %s", i, nome);
      i++;
   }
   return (0);
}
```

A Figura 21 ilustra o resultado da execução do programa, em que o laço é repetido até que a expressão relacional (i != 10) se torne falsa.

Anotações

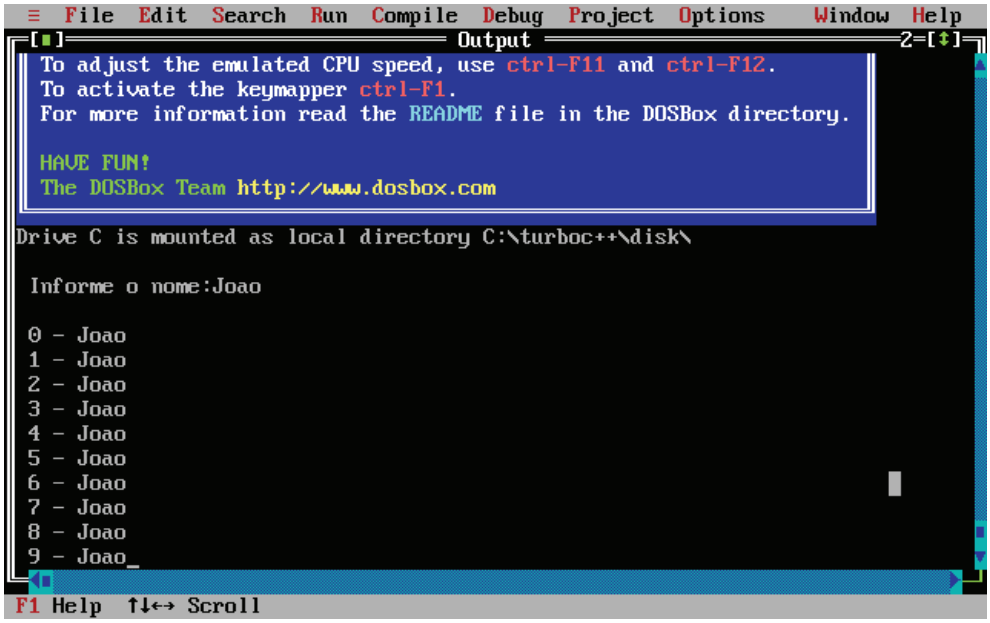


Figura 21 - Programa em C - Saída

Neste ponto você pode estar se questionando por que a variável i não foi inicializada com o valor um ao invés de zero. Você já sabe o porquê? Quando o valor de i é igual a 10 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 i != 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. Essa 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. Nesse tipo de estrutura, o trecho de código é executado pelo menos uma vez, pois o teste condicional é realizado no fim (ASCENCIO; CAMPOS, 2010).

Anotações

A sintaxe dessa 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 41 o programa para o problema descrito na seção anterior utilizando a estrutura **do while**.

Anotações

Quadro 41 - Programa em C

```
#include <stdio.h>
main ()
{
   char  nome[30];
   int   i ;
   printf("\n Informe o nome:");
   scanf("%s", nome);
   i = 0;
   do
   {
      printf("\n %d - %s", i, nome);
      i++;
   }
   while (i != 10);
   return (0);
}
```

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

PROBLEMA 1

O problema consiste em ler um conjunto de números inteiros e contar a quantidade de números pares e ímpares. A leitura dos números 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

Anotações

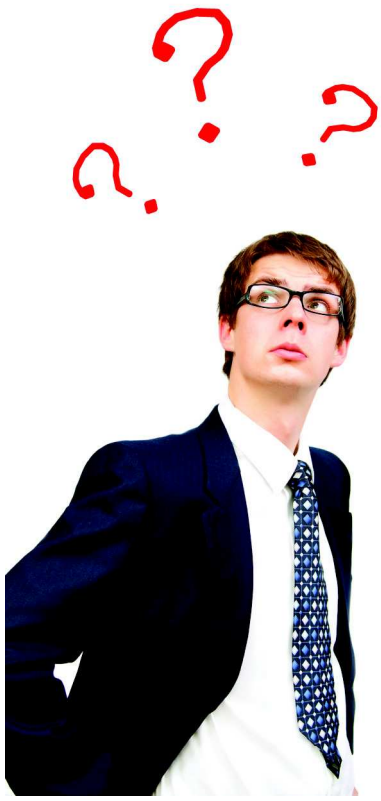
zero seja digitado. O processamento é contar a quantidade de números pares e ímpares. E a saída é informar quantos dos números lidos na entrada são pares e quantos são ímpares. No Quadro 42 é apresentado um programa para este problema utilizando a estrutura **while** e no Quadro 43 utilizando a estrutura **do while**.

Quadro 42 - Programa em C

```
#include <stdio.h>
main ()
{
   int   par, impar, num;
   par = 0;
   impar = 0;
   printf (“\n Informe o numero:”);
   scanf (“%d”, &num);
   while (num !=0)
   {
       if (num % 2 == 0)
           par++;
        else
           impar++;
     printf (“\n Informe o numero:”);
     scanf (“%d”, &num);
   }
   printf (“\n A quantidade de par e : %d”, par);
   printf (“\n A quantidade de impar e : %d”, impar);
   return (0);
}
```

Anotações

Fonte: SHUTTERSTOCK.COM



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? Nesse 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.

Anotações

Quadro 43 - Programa em C

```
#include <stdio.h>
main ()
{
   int   par, impar, num;
   par = 0;
   impar = 0;
   do
   {
      printf ("\n Informe o numero:");
      scanf ("%d", &num);
      if (num % 2 == 0)
          par++;
      else
           impar++;
    }
   while (num !=0);
   printf ("\n A quantidade de par e : %d", par);
   printf ("\n A quantidade de impar e : %d", impar);
   return (0);
}
```

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!! 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 aluno precisa obter média maior ou igual a 6. O programa é encerrado quando é informado o código de aluno 0.

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 44 temos a resolução do problema utilizando a estrutura **do while** e no Quadro 45 com a estrutura **while**.

Quadro 44 - Programa em C

```
#include <stdio.h>
main ()
{
   float  nota, soma, media;
   int    cod, i, nnota, naprovado, nreprovado;
   naprovado  = 0;
   nreprovado = 0;
   printf("\n Informe o numero de notas da disciplina:");
   scanf("%d", &nnota);
   do
   {
      printf ("\n Informe o codigo do aluno:");
      scanf ("%d", &cod);
      soma = 0;
      if (cod != 0)
      {
         for (i=1; i<=nnota; i++)
         {
            printf("Informe a %d nota do aluno:", i);
            scanf("%f", &nota);
            soma = soma + nota;
         }
      media = soma/nnota;
      if (media >=6)
         naprovado ++;
      else
        nreprovado ++;
      }
   }
   while (cod !=0);
   printf ("\n O numero de aprovados e : %d", naprovado);
   printf ("\n O numero de reprovados e : %d", nreprovado);
   return (0);
}
```

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.

Anotações

Quadro 45 - Programa em C

```
#include <stdio.h>
main ()
{
   float  nota, soma, media;
   int    cod, i, nnota, naprovado, nreprovado;
   naprovado  = 0;
   nreprovado = 0;
   printf("\n Informe o numero de notas da disciplina:");
   scanf("%d", &nnota);
   printf ("\n Informe o codigo do aluno:");
   scanf ("%d", &cod);
   while (cod != 0)
   {
       soma = 0;
       for (i=1; i<=nnota; i++)
       {
          printf("Informe a %d nota do aluno:", i);
          scanf("%f", &nota);
          soma = soma + nota;
       }
       media = soma/nnota;
       if (media >=6)
          naprovado ++;
       else
          nreprovado ++;
       printf ("\n Informe o codigo do aluno:");
       scanf ("%d", &cod);
   }
   printf ("\n O numero de aprovados e : %d", naprovado);
   printf ("\n O numero de reprovados e : %d", nreprovado);
   return (0);
}
```

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 estrutura 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 e OLIVEIRA, 1997; LOPES e GARCIA, 2002).

CONSIDERAÇÕES FINAIS

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

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

```
for ( i= valor inicial; condição;
incremento ou decremento de i);
{
    <instruções>;
}
```

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:

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

A estrutura **do while** é utilizada quando temos um número indefinido de repetições, no entanto, o teste lógico é realizado no final. A sintaxe desta estrutura é:

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

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 com a estrutura **do while**. Destaca-se que a estrutura **for** pode ser substituída pelo uso de estruturas baseadas em laços condicionais.

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.

Reflita

As estruturas **while** e **do while** podem ser substituídas uma pela outra, além de poderem substituir perfeitamente a estrutura **for**.

Saiba mais
sobre o **Assunto**

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

ATIVIDADE DE AUTOESTUDO

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. Construa um programa que leia o número de termos da série e imprima o valor de S.

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

3. Elabore um programa que imprima a tabuada do 1 ao 10.

4. Faça um programa que apresenta a soma de todos os números inteiros ímpares entre 200 e 500.

5. Construa um programa que apresente todos os números divisíveis por 3 e por 7 que sejam menores que 30.

6. 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, peso, sexo e o estado civil de várias pessoas e imprima a quantidade de pessoas 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 um determinado cômodo até que o nome do cômodo seja “FIM”. O programa deve apresentar o valor total acumulado da área residencial.

EXERCÍCIOS DE FIXAÇÃO

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

Quadro 46 - Programa em C

```
#include <stdio.h>
main ()
{
   int num,i, fat;
   printf (“\n Informe o numero:”);
   scanf (“%d”, &num);
   fat = 1;
   for (i=1; i <= num; i++)
       fat = fat * i;
   printf (“\n O fatorial e : %d”, fat);
   return (0);
}
```

2. Elabore um programa que apresente todos os números divisíveis por 3 que sejam menores que 100.

Quadro 47 - Programa em C

```
#include <stdio.h>
main ()
{
   int i;
   for (i=3; i<=100; i=i+3)
   {
      printf (“\n  %d”, i);
   }
   return (0);
}
```

3. Construa um programa que receba um número inteiro maior que 1 e verifique se ele é primo.

Quadro 48 - Programa em C

```
#include <stdio.h>
main ()
{
   int num, i, qtdade;
   printf("\n Informe o numero:" );
   scanf("%d", &num);
   qtdade = 0;
   for (i=1; i<=num; i++)
   {
      if (num % i == 0)

 qtdade ++;
   }
   if (qtdade == 2)
       printf ("\n O numero e primo.");
   else
       printf ("\n Nao e primo.");
   return (0);
}
```

4. A prefeitura 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.

Quadro 49 - Programa em C

```
#include <stdio.h>
main ()
{
   int filhos, npessoas;
   float salario, somas, somaf, msalario;
   npessoas = 0;
   somaf = 0;
   somas = 0;
   msalario = 0;
   printf (“\n Informe o salario:”);
   scanf (“%f”, &salario);
   printf (“\n Informe o numero de filhos:”);
   scanf (“%d”, &filhos);
   while (salario != -1)
   {
     npessoas++;
     if (salario > msalario)
        msalario = salario;
     somaf = somaf + filhos;
     somas = somas + salario;
     printf (“\n Informe o salario:”);
     scanf (“%f”, &salario);
     printf (“\n Informe o numero de filhos:”);
     scanf (“%d”, &filhos);
   }
   printf (“\n A media de salarios e : %.2f”, somas/npessoas);
   printf (“\n A media de filhos e : %.2f”, somaf/npessoas);
   printf (“\n O maior salario e : %.2f”, msalario);
   return (0);
}
```

5. Escreva um programa que receba a idade e 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 0 para a idade.

Quadro 50 - Programa em C

```
#include <stdio.h>
main ()
{
   int idade, npessoas;
   float altura, somaa, somai;
   char sexo;
   somaa = 0;
   somai = 0;
   npessoas = 0;
   printf (“\n Informe a idade:”);
   scanf (“%d”, &idade);
   printf (“\n Informe a altura:”);
   scanf (“%f”, &altura);
   while (idade != 0)
   {
      npessoas ++;
      somai = somai + idade;
      somaa = somaa + altura;
      printf (“\n Informe a idade:”);
      scanf (“%d”, &idade);
      printf (“\n Informe a altura:”);
      scanf (“%f”, &altura);
   }
   printf (“\n A media de altura e : %.2f”, somaa/npessoas);
   printf (“\n A media de idade e : %.2f”, somai/npessoas);
   return (0);
}
```

6. Em uma avaliação de um produto o cliente responde sua opinião (1 – satisfatório; 2 – indiferente; 3 – insatisfatório). Faça um programa que leia a idade e opinião e apresente: o número de clientes que respondeu satisfatório, a média de idade dos clientes que opinaram como satisfatório e o número de quem respondeu insatisfatório. O programa se encerra quando for digitado o valor 0 para idade.

Quadro 51 - Programa em C

```
#include <stdio.h>
main ()
{
   int idade, npessoas, npessoasi, npessoass, opiniao;
   float somai, media;
   npessoas = 0;
   npessoass = 0;
   npessoasi = 0;
   somai = 0;
   printf (“\n Informe a idade:”);
   scanf (“%d”, &idade);
   do
   {
      printf (“\n Informe a opiniao:”);
      scanf (“%d”, &opiniao);
   }
   while ((opiniao != 1) && (opiniao!=2) && (opiniao!=3));
   while (idade != 0)
   {
       npessoas ++;
       if (opiniao == 1)
    {
       somai= somai + idade;
       npessoass ++;
    }
    else
    {
       if (opiniao == 3)
       {
            npessoasi ++;
       }
    }
    printf (“\n Informe a idade:”);
    scanf (“%d”, &idade);
    do
    {
     printf (“\n Informe a opiniao:”);
     scanf (“%d”, &opiniao);
    }
    while ((opiniao != 1) && (opiniao!=2) && (opiniao!=3));
   }
   media = (somai/npessoass);
   printf (“\n O numero de pessoas insatisfeitas e : %d”, npessoasi);
   printf (“\n O numero de pessoas satisfeitas e: %d”, npessoass);
   printf (“\n A media de idade das pessoas satisfeitas e : %.2f”, media);
   return (0);
}
```

Leitura Complementar

OLIVEIRA, U. **Programando em C.** Editora Ciência Moderna, 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 ensinar a linguagem, ilustram algoritmos, estruturas de dados e técnicas de programação importantes. As principais rotinas de I/O são apresentadas e inclui um utilíssimo Manual de Referência C.

UNIDADE IV

VETORES, *STRINGS*, MATRIZES E ESTRUTURAS

Professora Me. Gislaine Camila Lapasini Leal

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*).

Plano de Estudo

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

- **Vetores**
- **Ordenação de Vetores**
- **Pesquisa em Vetores**
- ***Strings***
- **Matrizes**
- **Estruturas (*structs*)**
- **Problema**

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, sendo divididas em unidimensionais (vetores) e multidimensionais (matrizes). Veremos como atribuir valores, realizar a leitura e entrada de dados utilizando vetores e matrizes. Em relação aos vetores aprenderemos como realizar a classificação e a pesquisa por um determinado elemento.

Estudaremos as cadeias de caracteres (*strings*), como declarar e manipulá-las. Veremos as estruturas de dados heterogêneas, *structs*, que agregam informações de diversos tipos. Abordando, especificamente, como realizar atribuição, entrada, saída de dados e vetores de *structs*.

Ao término desta unidade você saberá construir programas utilizando vetores, matrizes, *strings* e *structs*. Entenderá, também, qual a importância desses conceitos e como utilizá-los em aplicações práticas.

VETORES

Um vetor consiste em um arranjo de elementos armazenados na memória principal, sequencialmente, todos com o mesmo nome. É um conjunto de variáveis de um mesmo tipo de dado as 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 nos colchetes representa o número de posições do vetor (LOPES; GARCIA, 2002).

A declaração de um vetor é dada por:

tipo_da_variável nome_da_variável [tamanho];

Um vetor de inteiros é 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 22. Esse 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 22 - Vetor

Fonte: SHUTTERSTOCK.COM

Fique atento!! Em C a numeração 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 índices associados ao identificador do vetor. Este índice pode ser referenciado diretamente ou por meio de uma

Anotações

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 essa verificação, o que pode ocasionar em erros durante a execução do programa.

No Quadro 52 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.

Quadro 52 - Programa em C

```
#include <stdio.h>
main ()
{
   int vetorA[10];
   int i;
   for (i=0;i<10;i++)
   {
      printf("\n Digite o %d elemento do vetor:", i);
      scanf("%d", &vetorA[i]);
   }
   printf ("\n Vetor preenchido");
   for (i=0;i<10;i++)
   {
      printf("\n O elemento na posicao %d e: %d", i, vetorA[i]);
   }
   return (0);
}
```

Anotações

No exemplo acima, observe que o inteiro i é inicializado em 0 tanto para a operação de atribuição quanto para a exibição dos elementos do vetor, pois a numeração de índices de um vetor sempre inicia em 0. Tanto a operação de leitura quanto a exibição é realizada por meio de um laço de repetição.

O resultado da execução do programa é apresentado na Figura 23. O que aconteceria se lêssemos mais de 10 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.

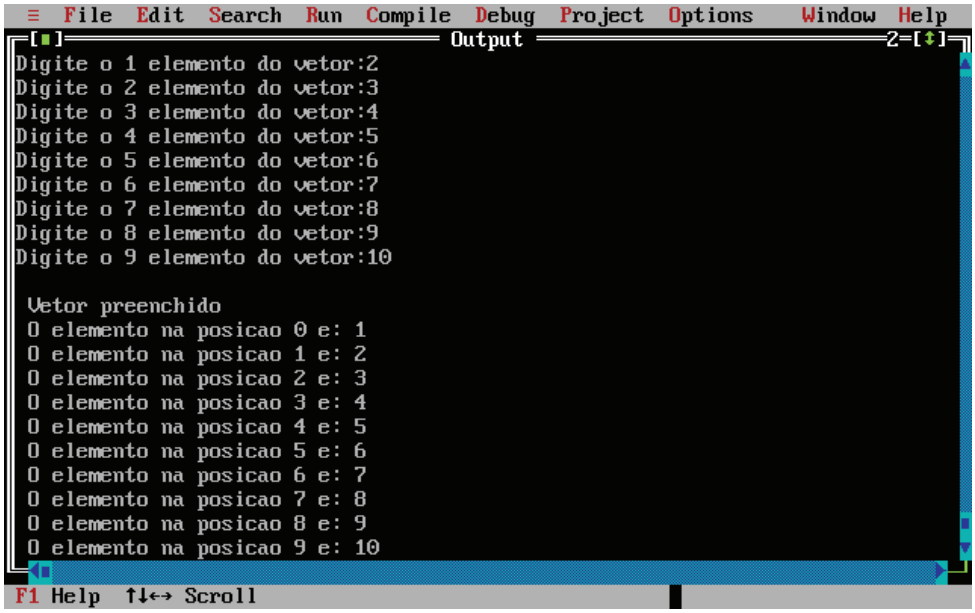


Figura 23 - Programa em C - Saída

Anotações

No Quadro 53 temos um exemplo de manipulação de vetores em que a leitura e a exibição extrapolam o valor definido para o vetor. 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.

Quadro 53 - Programa em C

```
#include <stdio.h>
main ()
{
   int vetorA[10];
   int i;
   for (i=0;i<15;i++)
   {
      printf("Digite o %d elemento:", i);
      scanf("%d", &vetorA[i]);
   }
   printf ("\n Vetor preenchido");
   for (i=0;i<15;i++)
   {
       printf("%d - ", vetorA[i]);
   }
   return (0);
}
```

Anotações

A Figura 24 ilustra o resultado da execução do programa (Quadro 53). 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.

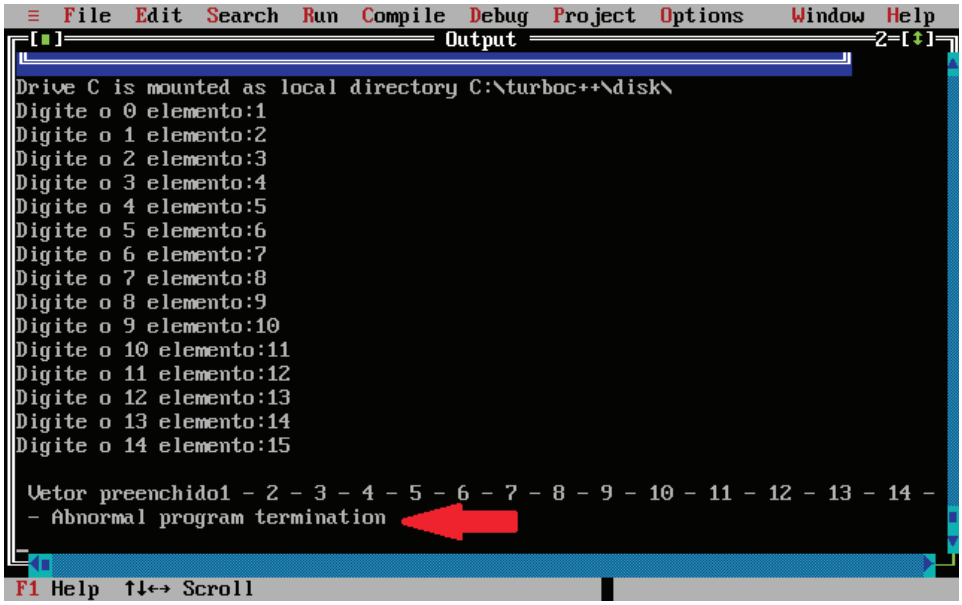


Figura 24 - Programa em C - Saída

Anotações

ORDENAÇÃO EM VETOR

Fonte: SHUTTERSTOCK.COM

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 (*Bubble Sort*) 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 54 temos um exemplo de ordenação utilizando o método da bolha. O primeiro laço de repetição (**for**) efetua a leitura dos dez elementos do vetor. Em seguida, temos dois laços de repetição (**for**) aninhados, os quais são responsáveis por percorrer o vetor comparando os elementos vizinhos (i e j) e se eles estão desordenados é efetuada a troca de posição. Por fim, o terceiro laço exibe os elementos do vetor já ordenados.

Anotações

Quadro 54 - Programa em C

```
#include <stdio.h>
main ()
{
   int vetorA[10];
   int i, j, troca;
   for (i=0;i<10;i++)
   {
    printf("Digite o %d elemento:", i);
    scanf("%d", &vetorA[i]);
   }
   for (i=0; i<9; i++)
   {
     for (j=i+1; j<10;j++)
     {
      if (vetorA[i] > vetorA[j])
      {
          troca = vetorA[i];
          vetorA[i] = vetorA[j];
          vetorA[j] = troca;
      }
    }
   }
  printf ("\n VETOR ORDENADO E \n");
  for (i=0;i<10;i++)
   {
    printf("%d -", vetorA[i]);
   }
  return (0);
}
```

A Figura 25 mostra a saída obtida na execução do programa (Quadro 54).

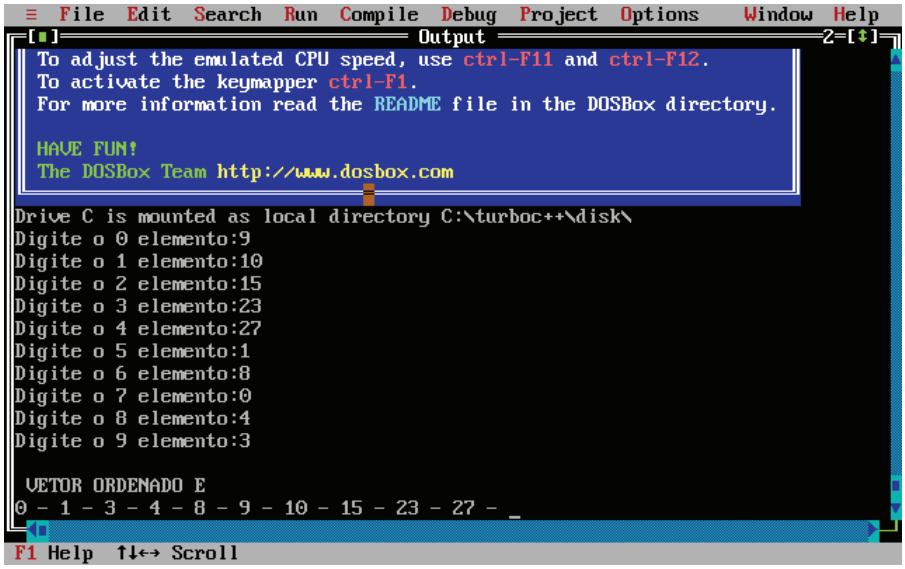


Figura 25 - Programa em C - Saída

PESQUISA EM VETOR

Em muitas situações temos que pesquisar por um 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, sequencialmente, até o último realizando testes lógicos verificando se o elemento do vetor, posição a posição, é igual ao elemento procurado. A pesquisa se encerra quando o elemento for encontrado ou quando todo o vetor foi percorrido e o elemento não foi encontrado (WIRTH, 1999).

Anotações

A implementação da pesquisa sequencial é apresentada no Quadro 55. Agora que você conhece o funcionamento e a implementação da pesquisa sequencial, execute o programa a seguir e analise o resultado de sua saída.

Anotações

Quadro 55 - Programa em C

```
#include <stdio.h>
main ()
{
   int vetorA[10];
   int i, acha, busca;
   for (i=0;i<10;i++)
   {
     printf("Digite o %d elemento:", i);
     scanf("%d", &vetorA[i]);
   }
   printf("Informe o elemento que deseja buscar:");
   scanf("%d", &busca);
   i = 0;
   acha = 0;
   while ((acha == 0) && (i < 10))
   {
     if (vetorA[i] == busca)
     {
       acha = 1;
     }
     else
     {
       i++;
     }
   }
   if (acha == 1)
     printf ("O elemento %d foi encontrado na posicao %d.", busca, i);
   else
     printf ("O elemento nao foi encontrado.");
   return (0);
}
```

STRINGS

A *string* consiste em uma cadeia de caracteres, um agregado de caracteres, 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. Além disso, devemos lembrar que as constantes *strings* aparecem entre aspas e não é necessário acrescentar o "\0", isso é realizado pelo compilador (ROCHA, 2006).

A linguagem C não possui um operador que atue com operandos do tipo *string*. Deste modo, a manipulação de *strings* é realizada por meio de funções. No Quadro 56 são apresentadas algumas funções para manipulação de *strings*.

Anotações

Quadro 56 - Funções para manipulação de *strings*

Função	Operação
strlen(x)	retorna o tamanho da string armazenada, sem '\0'.
strcat(x,y)	concatena na string x a string y, sem alterar y.
strcmp(x,y)	retorna a diferença em ASCII entre os dois primeiros caracteres diferentes, ou zero para igualdade.
strcpy(x,y)	copia uma string em outra, ou seja, copia y em x.
strncpy(x,y,n)	armazena em x os n primeiros caracteres de y.
strlwr(x)	retorna o minúsculo da string x.
strupr(x)	retorna o maiúsculo da string x.
strstr(x,y)	verifica se y é subcadeia da string x.
atoi(x)	converte x para o tipo int.
atol(x)	converte x para o tipo long.
atof(x)	converte x para o tipo float.
gets(x)	lê caracteres até encontrar o de nova linha ('\n') que é gerado quando se pressiona a tecla **[enter].**
puts(x)	imprime uma string de cada vez.

Fique tranquilo quanto ao modo de utilizar essas funções, veremos exemplos utilizando cada uma delas.

Anotações

O Quadro 57 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**.

Quadro 57- Programa em C

```
#include <stdio.h>
#include <string.h>
main ()
{
   char nome[80];
   int tamanho;
   printf (“\n Digite o seu nome completo:”);
   gets(nome);
   tamanho = strlen(nome);
   printf (“\n O comprimento do nome e: %d”, tamanho);
   return (0);
}
```

Anotações

A Figura 26 ilustra a saída obtida com a execução do programa (Quadro 55).

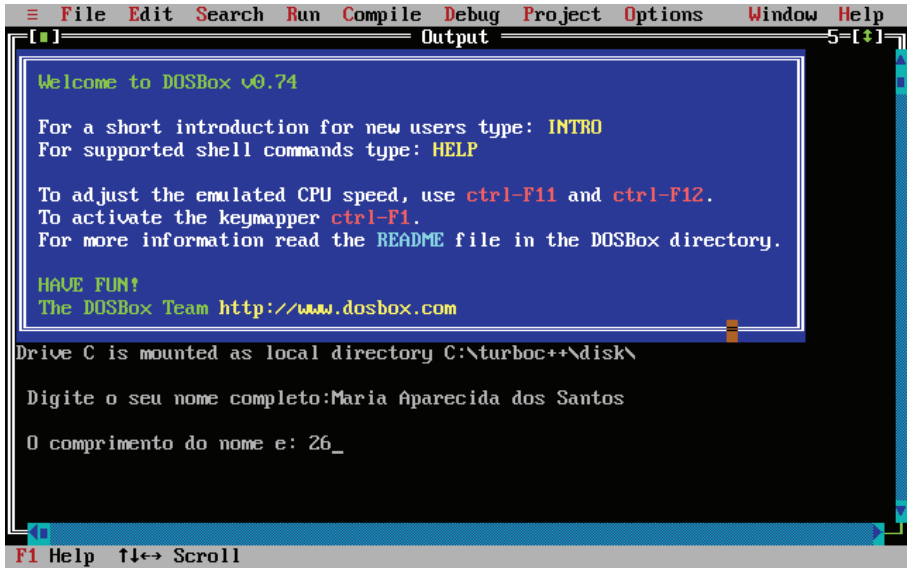


Figura 26 - Programa em C - Saída

Anotações

O programa do Quadro 58 demonstra o uso das funções **strcat**, **strstr**, **strupr** e **strlwr**.

Quadro 58 - Programa em C.

```
#include <stdio.h>
#include <string.h>
main ()
{
   char nome[80], sobrenome[80];
   printf ("\n Digite o seu nome:");
   gets(nome);
   printf ("\n Digite o seu sobrenome:");
   gets(sobrenome);
   strcat(nome, sobrenome);
   printf("\n Apos concatenar as strings temos que nome e %s:", nome);
   if (strstr(sobrenome, nome) == 0)
   {
        printf ("\n Agora o sobrenome esta contido na variavel nome");
   }
   strlwr(nome);
   printf("\n O nome em minusculo e: %s", nome);
   strupr(nome);
   printf("\n O nome em maiusculo e: %s", nome);
   return (0);
}
```

O resultado da execução do programa (Quadro 58) é mostrado na Figura 27. Ao concatenar as duas *strings* o nome e sobrenome ficaram juntos. O que devemos fazer para que fique com

Anotações

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, concatenar o sobrenome.

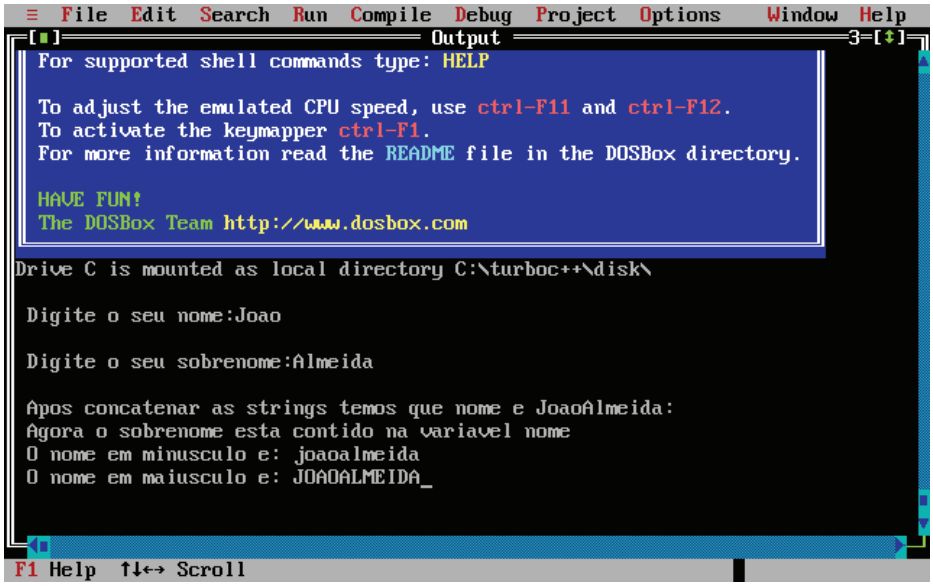


Figura 27 - Programa em C - Saída

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

MATRIZES

Uma matriz é um tipo de dado formado por uma sequência de variáveis do mesmo tipo, com o mesmo nome e alocadas sequencialmente na 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.

Anotações

A sintaxe para declaração de uma matriz é dada por:

```
tipo_da_variável nome_da_variável [tamanho][tamanho]...;
```

Como exemplo de declaração de matrizes 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 o número de repetições relativo ao tamanho das dimensões. Isto é, uma matriz de duas dimensões deve ser controlada por dois laços de repetição, de três dimensões três laços e assim por diante (MANZANO; OLIVEIRA, 1997).

O programa do Quadro 59 exemplifica a leitura e impressão de elementos de uma matriz.

Anotações

Quadro 59 - Programa em C

```
#include <stdio.h>
main ()
{
   int matrizA[2][10];
   int i, j;
   for (i=0;i<2;i++)
   {
     for (j=0;j<10;j++)
     {
       printf("\n Digite o %d %d elemento da matriz:", i, j);
       scanf("%d", &matrizA[i][j]);
     }
   }
   for (i=0;i<2;i++)
   {
     for (j=0;j<10;j++)
     {
       printf("\nO elemento da posicao %d %d e: %d", i, j, matrizA[i][j]);
     }
   }
   return (0);
}
```

Anotações

ESTRUTURAS

Ao utilizar vetores e matrizes conseguíamos manipular uma grande quantidade de dados por meio de um acesso indexado, no entanto tínhamos a limitação de que todos os elementos deveriam ser do mesmo tipo. Em diversas situações nos deparamos 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, os quais podem ser de diferentes tipos.

A sintaxe para declaração de uma estrutura é:

```
struct nome_da_estrutura
{
        tipo_de_dado do campo 1;
        tipo_de_dado do campo 2;
        . . .
        tipo_de_dado do campo n;
};
```

A partir da definição da estrutura o programa pode considerar que existe um novo tipo de dado a ser utilizado (nome_da_estrutura). Esse novo tipo de dado é capaz de armazenar informações que podem ser de tipos distintos (ASCENCIO; CAMPOS, 2010).

Anotações

Consideremos como exemplo a definição de uma estrutura que armazena a ficha de um produto, conforme ilustra a Figura 28. 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.

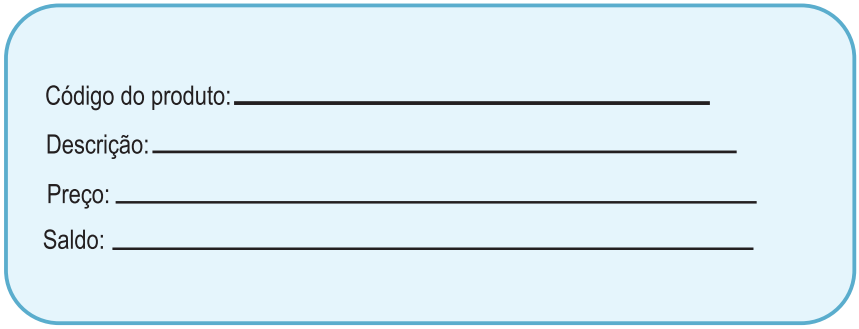


Figura 28 - Ficha de produto

A declaração da estrutura para a ficha do produto é:

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

Essa declaração indica que o programa poderá utilizar um novo tipo de dado que contém quatro informações. 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

Anotações

a *struct* seja acessível de qualquer parte do programa temos que defini-la fora da função *main*, preferencialmente após os *includes*.

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

nome_da_estrutura nome_da_variável;

No caso da ficha de produto teríamos a seguinte declaração:

produto ficha;

Essa declaração nos indica que temos uma variável denominada ficha que é do tipo 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 60 temos um programa que exemplifica as operações leitura e escrita utilizando *struct*. Observe que inicialmente temos a declaração da *struct* após a diretiva *include* e no interior da função *main* temos a declaração de uma variável do tipo 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.

Anotações

Quadro 60 - Programa em C

```
#include <stdio.h>
struct produto
{
   int   codigo;
   char  descricao[50];
   float preco;
   int   saldo;
};
main ()
{
   produto ficha;
   printf("\n Digite o codigo do produto:");
   scanf("%d", &ficha.codigo);
   printf("\n Digite a descricao do produto:");
   scanf("%s", ficha.descricao);
   printf("\n Digite o preco do produto:");
   scanf("%f", &ficha.preco);
   printf("\n Digite o saldo do produto:");
   scanf("%d", &ficha.saldo);
   printf("\n Codigo : %d", ficha.codigo);
   printf("\n Descricao : %s", ficha.descricao);
   printf("\n Preco : %.2f", ficha.preco);
   printf("\n Saldo : %d", ficha.saldo);
   return (0);
}
```

Anotações

Fonte: SHUTTERSTOCK.COM

O exemplo anterior ilustra a leitura e escrita da ficha de um produto. E se quisermos armazenar a ficha de 10 produtos, precisamos criar 10 variáveis do tipo produto? Não, podemos criar um vetor de *struct*.

O Quadro 61 exemplifica o uso de um vetor de *struct* com operações de leitura e escrita.

Anotações

Quadro 61 - Programa em C

```
#include <stdio.h>
#include <iostream.h>
struct produto
{
  int   codigo;
  char  descricao[50];
  float preco;
  int   saldo;
};
main ()
{
    produto ficha [10];
    int i;
    for (i=0; i<10; i++)
    {
      printf("\n Digite o codigo do produto:");
      cin >> ficha[i].codigo;
      printf("\n Digite a descricao do produto:");
      gets(ficha[i].descricao);
      printf("\n Digite o preco do produto:");
      cin >> ficha[i].preco;
      printf("\n Digite o saldo do produto:");
      cin >> ficha[i].saldo;
    }
    for (i=0; i<10;i++)
    {
      printf ("\n PRODUTO %d", i+1);
      printf(" Codigo : %d", ficha[i].codigo);
      printf(" Descricao : %s", ficha[i].descricao);
      printf(" Preco : %.2f", ficha[i].preco);
      printf(" Saldo : %d", ficha[i].saldo);
    }
    return (0);
}
```

PROBLEMA

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 62 temos o programa para o problema acima 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.

Quadro 62 - Programa em C

```
#include <stdio.h>
#include <iostream.h>
#include <conio.h>
struct livro
{
  int   codigo;
  char  titulo[50];
  char  autor[30];
  char  area[30];
  int   ano;
  char  editora[30];
};
main ()
{
    livro ficha [20];
    livro troca;
    int busca, i, j, acha, op;
    op = 0;
    while  (op !=5)
    {
      printf("\n 1 - Cadastrar os livros");
      printf("\n 2 - Imprimir os livros cadastrados");
      printf("\n 3 - Pesquisar livros por area");
      printf("\n 4 - Ordenar os livros por ano");
      printf("\n 5 - Sair");
      printf("\n Digite a opcao desejada: ");
      scanf("%d", &op);
      if (op == 1)
      {
          clrscr();
          for (i=0; i<20; i++)
```

```
        {
            printf("Digite o codigo do livro %d:", i+1 );
            cin >> ficha[i].codigo;
            printf("Digite o titulo do livro: ");
            cin >> ficha[i].titulo;
            printf("Digite o nome do autor: ");
            cin >> ficha[i].autor;
            printf("Digite a area do livro: ");
            cin >> ficha[i].area;
            printf("Digite o ano : ");
            cin >> ficha[i].ano;
            printf("Digite o nome da editora: ");
            cin >> ficha[i].editora;
      }
   }
   else
   {
        if (op == 2)
        {
            clrscr();
           for (i=0; i<20; i++)
           {
               printf("\n CODIGO: %d", ficha[i].codigo );
               printf("\n TITULO: %s", ficha[i].titulo);
               printf("\n AUTOR: %s", ficha[i].autor);
               printf("\n AREA: %s", ficha[i].area);
              printf("\n ANO: %d", ficha[i].ano);
              printf("\n EDITORA: %s", ficha[i].editora);
           }
   }
   else
   {
      if (op == 3)
```

```
{
    clrscr();
    printf("Digite o codigo que deseja buscar :");
    scanf ("%d", &busca);
    i = 0;
   acha =0;
    while ((i<20) && (acha == 0))
    {
        if (ficha[i].codigo == busca)
            acha = 1;
        else
            i++;
    }
   if (acha ==1)
    {
       printf("\n CODIGO: %d", ficha[i].codigo );
       printf("\n TITULO: %s", ficha[i].titulo);
       printf("\n AUTOR: %s", ficha[i].autor);
       printf("\n AREA: %s", ficha[i].area);
       printf("\n ANO: %d", ficha[i].ano);
       printf("\n EDITORA: %s", ficha[i].editora);
    }
  else
       printf("\n Registro nao encontrado");
 }
else
{
      if (op ==4)
      {
        clrscr();
        for (i=0;i<19;i++)
        {
                for (j=i+1;j<20;j++)
```

```
                    {
                      if (ficha[i].ano > ficha[j].ano)
                      {
                        troca = ficha[i];
                        ficha[i]= ficha[j];
                        ficha[j] = troca;
                      }
                    }
                }
                for (i=0; i<20;i++)
                {
                  printf("\n CODIGO: %d, TITULO: %s, ANO: %d", ficha[i].
codigo, ficha[i].titulo, ficha[i].ano);
                }
            }
        }
      }
    }
  return (0);
}
```

Observe que utilizamos a função **clrscr**(). Essa função é responsável por limpar o texto corrente da tela. Para utilizá-la em nossos programas temos que dar um include no arquivo conio.h.

É importante que você execute este programa e analise o seu funcionamento, pois na construção do programa utilizamos todos os conceitos vistos no decorrer desta unidade. Se você ainda ficou com dúvidas, analise os exercícios de fixação.

Anotações

CONSIDERAÇÕES FINAIS

Nesta unidade você aprendeu a construir programa 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 *Bubble Sort,* que varre o vetor repetidas vezes, comparando os elementos vizinhos. Se eles estão fora de ordem, é efetuada uma troca de posição. Esse 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 conhecemos funções que permitem concatenar, comparar, copiar, armazenar, imprimir, converter para um valor numérico e 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 de estruturas.

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.

Saiba mais
sobre o **Assunto**

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=rCKbfdvnhxI&feature=related>.

Para saber um pouco mais sobre a declaração e construção de *structs,* acesse: <http://pt.wikibooks.org/wiki/Programar_em_C/Estruturas>.

Reflita

As constantes *strings* aparecem sempre entre aspas e devemos acrescentar o caractere de fim de *string* "\0" (ROCHA, 2006).

ATIVIDADE DE AUTOESTUDO

1. Faça um programa que leia dois vetores A e B e apresente a diferença entre os elementos.
2. Escreva um programa que leia um vetor A e o apresente em ordem decrescente.
3. Elabore um programa que leia uma palavra e se a palavra tiver número ímpar de caracteres, imprima todas as suas consoantes.
4. Faça um programa que leia uma palavra e o número de vezes que se deseja imprimi-la.
5. Construa um programa que recebe duas matrizes inteiras de ordem 5 e imprima a soma e a diferença entre elas.
6. Faça um programa que efetua a leitura dos nomes de 5 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. 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. 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.

EXERCÍCIOS DE FIXAÇÃO

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

Quadro 63 - Programa em C

```
#include <stdio.h>
main ()
{
   int vetorA[30];
   int i;
   for (i=0;i<30;i++)
   {
      printf("Digite o %d elemento:", i);
      scanf("%d", &vetorA[i]);
   }
   for (i=29;i>=0;i--)
   {
      printf("\n %d", vetorA[i]);
   }
   return (0);
}
```

2. 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.

Quadro 64 - Programa em C

```
#include <stdio.h>
main ()
{
   int vetorA[30], vetorB[30], vetorC[30];
   int i, j, troca;
   for (i=0;i<30;i++)
   {
     printf("Digite o %d elemento do vetor A:", i);
     scanf("%d", &vetorA[i]);
     printf("Digite o %d elemento do vetor B:", i);
     scanf("%d", &vetorB[i]);
     vetorC[i] = vetorA[i] + vetorB[i];
   }
   for (i=0;i<29;i++)
   {
     for (j=i+1; j<30; j++)
     {
          if (vetorC[i] > vetorC[j])
          {
               troca = vetorC[i];
               vetorC[i] = vetorC[j];
               vetorC[j] = troca;
          }
     }
   }
   for (i=0;i<30;i++)
   {
      printf("%d - ", vetorC[i]);
   }
   return (0);
}
```

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

Quadro 65 - Programa em C

```
#include <stdio.h>
#include <string.h>
main ()
{
   char nome[30];
   int tam, i;
   printf ("\n Digite o nome:");
   gets(nome);
   tam = strlen(nome);
   for (i=0;i<tam; i++)
   {
    if (i % 2 == 0)
      printf ("\n %c", nome[i]);
   }
   return (0);
}
```

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

Quadro 66 - Programa em C

```
#include <stdio.h>
#include <string.h>
main ()
{
   char nome[30];
   int tam, i;
   printf (“\n Digite o nome:”);
   gets(nome);
   tam = strlen(nome);
   for (i=tam;i>=0; i--)
   {
     printf (“%c”, nome[i]);
   }
   return (0);
}
```

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

Quadro 67 - Programa em C

```
#include <stdio.h>
#include <string.h>
main ()
{
   char nome[30];
   int tam, i;
   printf (“\n Digite o nome:”);
   gets(nome);
   tam = strlen(nome);
   for (i=0;i<tam; i++)
   {
      printf (“\n %s”, nome);
   }
   return (0);
}
```

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

Quadro 68 - Programa em C

```
#include <stdio.h>
main ()
{
   float media[20];
   float notas[20] [4];
   float somat, mediat, soma;
   int   i, j;
   somat = 0;
   mediat =0;
   for (i=0;i<20;i++)
   {
    soma = 0;
    for (j=0; j<4; j++)
    {
      printf("Informe a nota %d do aluno %d:", j+1, i+1);
      scanf("%f", &notas[i][j]);
      soma = soma + notas[i][j];
    }
    media[i] = soma/4;
    somat= somat + media[i];
   }
   mediat = somat/20;
   for (i=0; i<20;i++)
   {
    printf("\n A media do aluno %d e %.2f:", i, media[i]);
   }
   printf ("\n A media da turma e : %.2f", mediat);
   return (0);
}
```

7. 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 é apresentado o setor e o salário. Se a matrícula digitada não existir, informar o usuário.

Quadro 69 - Programa em C

```
#include <stdio.h>
#include <iostream.h>
struct funcionario
{
 int   matricula;
 char  nome[50];
 char  setor[30];
 float salario;
};
main ()
{
   funcionario ficha [20];
   int busca, i, acha;
   char op;
   for (i=0; i<20; i++)
   {
    printf("\n Digite a matricula do funcionario:");
    cin >> ficha[i].matricula;
    printf("\n Digite o nome:");
    gets(ficha[i].nome);
    printf("\n Digite o setor:");
    cin >> ficha[i].setor;
    printf("\n Digite o salario:");
    cin >> ficha[i].salario;
   }
  do
   {
    printf("Deseja realizar busca (S/N):");
    cin >> op;
   }
```

```
   while ((op != 'S') && (op!='s') && (op !='n') && (op !='N'));
   while ((op=='S') || (op=='s'))
   {
    printf("\n Informe a matricula que deseja buscar:");
    cin >> busca;
    i = 0;
    acha = 0;
    while ((i<20) && (acha==0))
    {
     if (ficha[i].matricula == busca)

acha = 1;
      else

i++;
    }
    if (acha ==1)
    {
     printf ("\n O setor e: %s", ficha[i].setor);
     printf ("\n O salario e: %.2f", ficha[i].salario);
    }
    else
    {
     printf ("\n Matricula nao cadastrada");
    }
    do
    {
     printf("Deseja realizar busca (S/N):");
     cin >> op;
    }
    while ((op != 'S') && (op!='s') && (op !='n') && (op !='N'));
   }
   return (0);
}
```

Leitura Complementar

HERBERT, S. **C Completo e Total**. Editora Makron, 3. ed, 1997.

Sinopse: "C Completo e Total", 3ª 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, 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.

UNIDADE V

FUNÇÕES E ARQUIVOS

Professora Me. Gislaine Camila Lapasini Leal

Objetivos de Aprendizagem

- Conhecer e desenvolver funções.
- Estudar a passagem de parâmetros por valor e por referência.
- Entender o conceito de recursividade.
- Trabalhar com manipulação de arquivos em C.

Plano de Estudo

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

- **Funções**
- **Escopo de Variáveis**
- **Passagem de Parâmetros**
- **Passagem de Parâmetros por Valor**
- **Passagem de Parâmetros por Referência**
- **Protótipo de Funções**
- **Recursividade**
- **Arquivos**

INTRODUÇÃO

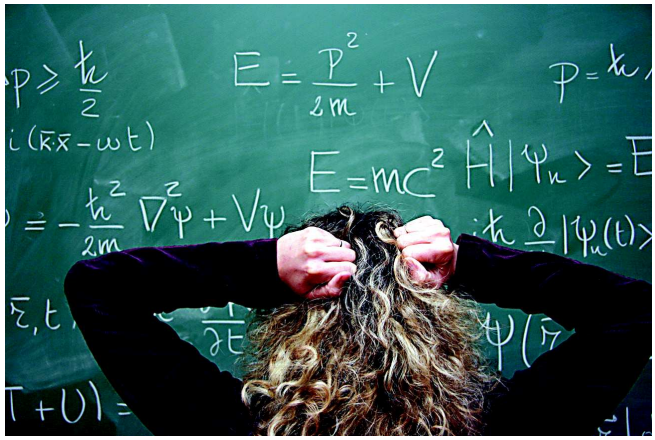
Chegamos à última unidade da disciplina de Algoritmos e Lógica de Programação II. Nesta unidade, você aprenderá a construir programas utilizando o conceito de modularização e a manipular arquivos.

Para aplicar o conceito de modularização estudaremos a construção de funções na linguagem C, destacando sua sintaxe e modo de aplicação. Ao trabalhar com funções precisaremos entender o conceito de escopo de variáveis, compreender o que são variáveis locais e variáveis globais e como elas influenciam nossos programas.

Estudaremos como construir funções que recebem parâmetros e como podemos realizar a passagem desses parâmetros (por valor ou por referência). Veremos como construir funções recursivas. Conheceremos o conceito de arquivos e como manipulá-los utilizando operações de 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?

FUNÇÕES



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

No particionamento dos problemas complexos utilizamos sub-rotinas para resolver cada subproblema, permitindo assim a modularização. A linguagem C possibilita a modularização por meio de funções.

Uma função é uma sub-rotina 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 e ou até mesmo outras funções.

O objetivo de uma função é realizar alguma subtarefa específica, isto é, podemos escrever

Anotações

funções para a entrada de dados, processamento e saída de dados.

A sintaxe de uma função é dada por:

```
tipo_de_retorno nome_da_função (declaração_de_parâmetros)
{
      corpo_da_função
}
```

Em que:

- tipo_de_retorno: indica o tipo de variável que a função retornará.
- declaração_de_parâmetros: especificação das variáveis de entrada da função. Devemos especificar o tipo de cada uma das variáveis.
- 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 função e retornar o valor informado. Lembre-se: o valor informado no comando *return* deve ser compatível com o tipo declarado para a função.

No Quadro 70 temos um programa que utiliza uma função denominada soma() para efetuar a 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, os 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.

Anotações

Quadro 70 - Programa em C

```
#include <stdio.h>
#include <conio.h>
int soma ()
{
  float num1, num2, total;
  printf("Digite o primeiro numero :");
  scanf ("%f", &num1);
  printf("Digite o segundo numero :");
  scanf ("%f", &num2);
  total = num1+num2;
  printf ("A soma e %.2f", total);
  return (0);
}
main ()
{
  clrscr();
  soma();
  return (0);
}
```

A linguagem C possui o tipo void. Esse tipo quer dizer vazio 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:

void nome_da_função (declaração_de_parâmetros)

Anotações

Um exemplo utilizando o tipo void é apresentado no Quadro 71. Observe que em funções do tipo **void** não utilizamos o comando *return*, pois não temos valor de retorno. Além disso, note que inserimos o tipo *void* para a função *main*.

Quadro 71 - Programa em C

```
#include <stdio.h>
#include <conio.h>
void soma ()
{
  float num1, num2, total;
  printf("Digite o primeiro numero :");
  scanf ("%f", &num1);
  printf("Digite o segundo numero :");
  scanf ("%f", &num2);
  total = num1+num2;
  printf ("A soma e %.2f", total);
}
void main ()
{
  clrscr();
  soma();
}
```

Na seção seguinte estudaremos o escopo das variáveis.

Anotações

ESCOPO DE VARIÁVEIS

O escopo de uma variável é relacionado a sua visibilidade em relação às sub-rotinas 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).

Fonte: SHUTTERSTOCK.COM

O que acontece se declararmos todas as variáveis como global? Esse 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 uso desse tipo de variável.

Anotações

PASSAGEM DE PARÂMETROS

As funções com passagem de parâmetros são aquelas que recebem valores no momento em que são chamadas. O Quadro 72 apresenta um programa utilizando a passagem de parâmetros.

A função soma é do tipo **float** e possui dois parâmetros n1 e n2, os quais também são do tipo **float**. Note que a entrada de dados é realizada na função *main*, em que temos as operações de leitura. Após a leitura é realizada uma chamada à função soma, passando como parâmetros num1 e num2. A função soma realiza a operação de adição dos dois valores obtidos como parâmetro e retorna o total.

Como a função soma retorna um valor do tipo **float** sua chamada foi realizada diretamente em uma instrução de atribuição. Isto significa que o valor retornado pela função é atribuído diretamente para a variável resposta.

Quadro 72 - Programa em C

```
#include <stdio.h>
#include <conio.h>
float soma (float n1, float n2)
{
  float total;
  total = n1+ n2;
  return (total);
}
void main ()
{
  float num1, num2, resposta;
  clrscr();
  printf("Digite o primeiro numero :");
  scanf ("%f", &num1);
  printf("Digite o segundo numero :");
  scanf ("%f", &num2);
  resposta = soma(num1, num2);
  printf ("A soma e %.2f", resposta);
}
```

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. Os parâmetros reais são os valores obtidos na entrada de dados e os formais são os parâmetros exigidos na função. Com isto, 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

Anotações

pelos reais no momento da execução da função. A passagem de parâmetros pode ser realizada 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 valor 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 chamada (MANZANO e OLIVEIRA, 1997; ASCENCIO e CAMPOS, 2010).

No Quadro 73 é apresentado um programa em C que recebe um número inteiro, calcula o seu dobro e exibe.

Quadro 73 - Programa em C

```
#include <stdio.h>
#include <conio.h>
int calcdobro (int x)
{
  x = x * x;
  return x;
}
void main ()
{
  int x, resposta;
  clrscr();
  printf("Digite um numero inteiro :");
  scanf ("%d", &x);
  resposta = calcdobro(x);
  printf ("O dobro do numero %d e %d", x, resposta);
}
```

O parâmetro formal da função calcdobro sofre alterações no interior da função, mas a variável x da função main permanece inalterada, como pode ser visto na Figura 29.

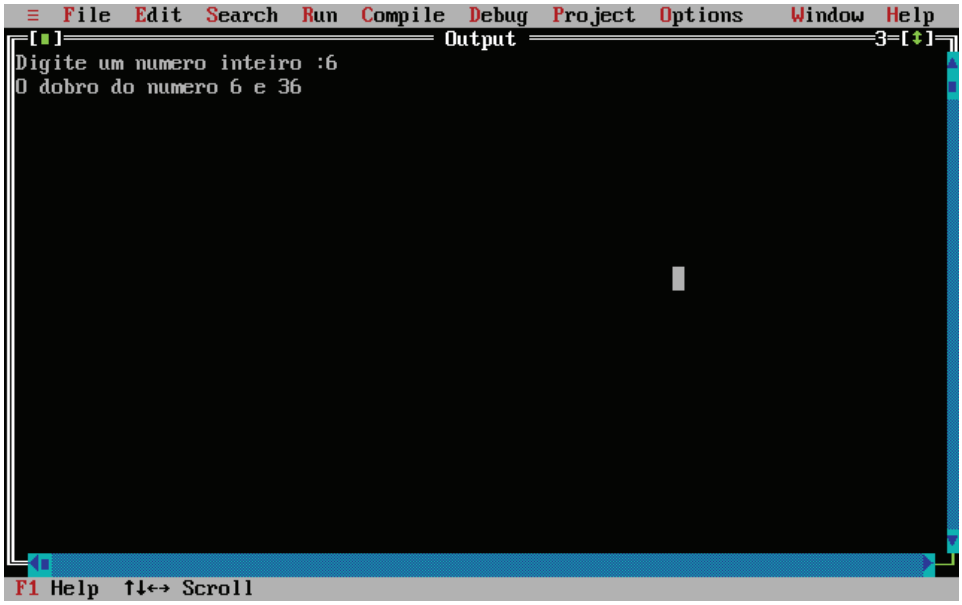


Figura 29 - Programa em C - Saída

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 74.

Anotações

Quadro 74 - Programa em C

```
#include <stdio.h>
#include <conio.h>
int calcdobro (int *x)
{
  *x = *x * (*x);
  return *x;
}
void main ()
{
  int x, resposta;
  clrscr();
  printf("Digite um numero inteiro :");
  scanf ("%d", &x);
  resposta = calcdobro(&x);
  printf ("O dobro do numero %d e %d", x, resposta);
}
```

Fonte: SHUTTERSTOCK.COM

Anotações

Observe que na chamada da função *calcdobro* temos que passar o endereço de memória ocupado pela variável x. 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, deste modo temos que inserir o caractere * antes do nome das variáveis. E ao final, o que muda em relação à execução?

A Figura 30 ilustra a saída obtida com a execução do programa. Podemos observar que ao utilizar a passagem de parâmetros por referência temos a alteração do valor da variável x, que inicialmente tinha o valor 5 e ao final da execução da função *calcdobro* ficou com o valor 25. Por que isto acontece?

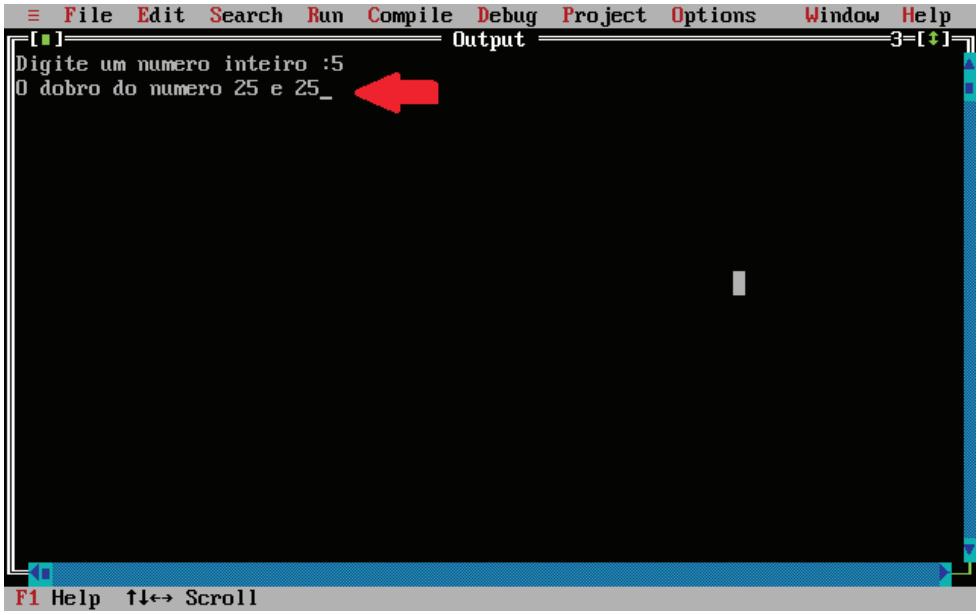


Figura 30 - Programa em C - Saída

Por que não temos uma cópia do valor de x na passagem de parâmetros e sim o seu endereço de memória. Com isto, temos que a alteração realizada é sobre x, conforme Figura 30.

Anotações

PROTÓTIPO DE FUNÇÕES

Fonte: SHUTTERSTOCK.COM

Você observou que até o momento todas as nossas funções foram escritas antes da função *main*? Podemos inserir nossas funções após a função *main*? Em qualquer programa podemos escrever as funções antes ou após a função *main*. No entanto, se optarmos por escrevê-las depois da função *main* temos que utilizar 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 (declaração_de_parâmetros);
```

Anotações

O programa da Figura 31 apresenta um exemplo de programa com uma função escrita depois da função *main*. Para que o compilador reconheça esta função temos que aplicar o conceito de protótipo de uma função, em que declaramos o cabeçalho da função *calcdobro* seguido de ponto e vírgula antes da função *main* (conforme Figura 31).

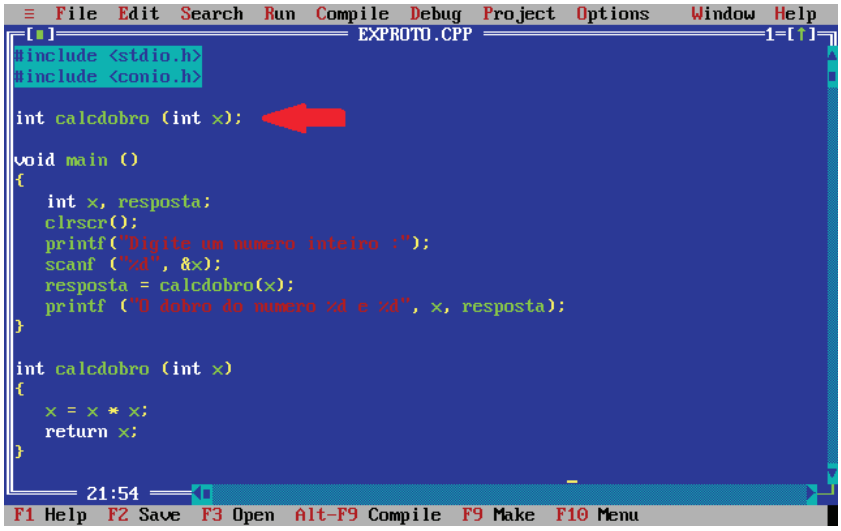


Figura 31 - Programa em C

RECURSIVIDADE

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 irá parar de fazer chamadas a si mesmo, impedindo que entre em um *loop*.

Anotações

No Quadro 75 temos a implementação da função recursiva para cálculo do fatorial utilizando a linguagem C.

Quadro 75 - Programa em C

```
#include <stdio.h>
#include <conio.h>
int fatorial (int x)
{
  if (x == 0)
    return 1;
  else
    return x*(fatorial(x-1);
}
void main ()
{
  int num, resposta;
  clrscr();
  printf("Digite um numero inteiro :");
  scanf ("%d", &num);
  resposta = fatorial(num);
  printf ("O fatorial e %d", resposta);
}
```

Anotações

O Quadro 76 apresenta a função recursiva para o cálculo da série de Fibonacci.

Quadro 76 - Programa em C

```
#include <stdio.h>
#include <conio.h>
int fibonacci (int x)
{
  if ((x == 0) || (x==1))
    return x;
  else
    return fibonacci(x-2) + fibonacci(x-1);
}
void main ()
{
  int num, resposta;
  clrscr();
  printf("Digite um numero inteiro :");
  scanf ("%d", &num);
  resposta = fibonacci(num);
  printf ("O fibonacci e %d", resposta);
}
```

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. Essa 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:

```
FILE *arq, *pont;
```

Para abrir um arquivo utilizamos a função fopen(), a qual abre um arquivo e retorna o ponteiro associado a ele. Sua sintaxe é:

```
FILE *p;

p = fopen(nome_do_arquivo, modo_de_abertura);
```

Em que:

- p: variável que armazena o endereço inicial de memória ocupado por um arquivo;
- nome_do_arquivo: nome do arquivo que se deseja abrir;
- modo_da_abertura: representa como o arquivo será aberto.

Quadro 77 - Modos para abertura de arquivos

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

Fonte: adaptado de (ASCENCIO; CAMPOS, 2010)

A função **fopen**() quando utilizada no modo escrita, cria o arquivo especificado. Se esse não existir ou se já existe um arquivo com o mesmo nome, será sobreposto por um novo arquivo vazio. O resultado da função **fopen**() é o endereço de memória ocupado pelo arquivo ou NULL quando ocorre algum erro e o arquivo não é aberto (ROCHA, 2006; ASCENCIO e 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 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 uma arquivo é realizada por meio da função **fputc**(), que possui a seguinte sintaxe:

fputc(char ch, FILE *arq);

Anotações

em que:

- ch: corresponde ao caractere 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:

int **fgetc**(FILE *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(char *cadeia, FILE *arq);

O argumento cadeia armazena a cadeia de caracteres que será escrito 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(char *cadeia, int tam, FILE *arq);

Onde:

- cadeia: armazena a cadeia de caracteres obtida do arquivo;
- tam: indica que a quantidade máxima de caracteres lidos será tam-1;
- 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(void *mem, size_t qtd_bytes, size_t cont, FILE *arq);

Em que:

- mem: representa a variável que armazena o conteúdo a ser gravado;
- 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 caractere de um arquivo utilizamos a função **fread**(), que apresenta a seguinte sintaxe:

fread(void *mem, size_t qtd_bytes, size_t cont, FILE * arq);

Onde:

- mem: representa a variável que receberá o conteúdo lido;
- 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, a 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.

Anotações

No Quadro 78 são apresentadas algumas funções para manipulação de arquivos.

Quadro 78 - Funções para manipulação de arquivos

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

Fique tranquilo!!! Veremos o funcionamento destas funções. O Quadro 79 apresenta um programa que exemplifica a manipulação de arquivos. É realizada a operação de abertura do arquivo, verificação de erro, leitura de um caractere e fechamento de arquivo. A leitura de caractere é realizada até que seja digitado o caractere 'f'. Execute o código a seguir e ao final, abra o arquivo com nome "arquivo.txt" e verifique o seu conteúdo.

Anotações

Quadro 79 - Programa em C

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
void main()
{
  FILE *arq;
  char letra;
  arq = fopen("C:\\arquivo.txt", "w");
  if (arq == NULL)
  {
     printf ("\n O arquivo nao foi aberto. Ocorreu um erro!");
  }
  else
  {
     printf ("\n Digite um caractere: ");
     cin >> letra;
     while ((letra != 'f') && (letra !='F'))
     {

fputc(letra, arq);

if (ferror(arq))

{
            printf("\n Erro na gravacao !!");
}

else

{
            printf ("\n Gravacao efetuada com sucesso!! ");
}

printf ("\n Digite outro caractere: ");

cin >> letra;
     }
  }
  fclose(arq);
}
```

No Quadro 80 temos um exemplo de manipulação de arquivo com a gravação de cadeias de caracteres até que seja informada a palavra “fim”.

Quadro 80 - Programa em C

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <string.h>
void main()
{
  FILE *arq;
  char palavra[50];
  arq = fopen(“C:\\arquivo.txt”, “w”);
  if (arq == NULL)
  {
     printf (“\n O arquivo nao foi aberto. Ocorreu um erro!”);
  }
  else
  {
     printf (“\n Digite uma palavra: ”);
     gets(palavra);
     while ((stricmp(palavra,“fim”) != 0))
     {

fputs(palavra, arq);

if (ferror(arq))

{
          printf(“\n Erro na gravacao !!”);
}
```

Anotações

```
else

{
            printf (“\n Gravacao efetuada com sucesso!! ”);
}

printf (“\n Digite uma palavra: ”);

gets(palavra);
      }
  }
  fclose(arq);
}
```

Um exemplo de operação de leitura de um arquivo texto é apresentado no Quadro 81. A leitura do arquivo é realizada até que seja encontrado o fim do arquivo. De modo que são efetuadas leituras com 50 caracteres.

Anotações

Quadro 81 - Programa em C

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <string.h>
void main()
{
  FILE *arq;
  char frase[50];
  arq = fopen("C:\\teste.txt", "r");
  if (arq == NULL)
  {
     printf ("\n O arquivo nao foi aberto. Ocorreu um erro!");
  }
  else
  {
    while (!feof(arq))
    {
      fgets(frase, 50, arq);
      if (ferror(arq))
      {
          printf("Erro na leitura do arquivo");
      }
      else
      {
          printf("\n Leitura realizada com sucesso. A cadeia e : %s ",
frase);
      }
    }
  }
  fclose(arq);
}
```

Observe que quando queremos realizar a leitura ou 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 79 e Quadro 80 a abertura do arquivo foi realizada utilizando o modo de acesso de gravação (w) e no Quadro 81 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 77.

CONSIDERAÇÕES FINAIS

Nesta unidade você aprendeu como modularizar os programas utilizando funções. Vimos que as funções são sub-rotinas que nos permitem desviar a execução do programa principal para realizar uma tarefa específica e 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 e/ou até mesmo outras funções.

Aprendemos que a sintaxe de uma função é dada por:

```
tipo_de_retorno nome_da_função (declaração_de_parâmetros)
{
corpo_da_função
}
```

Ainda em relação a funções vimos o uso do comando *return*, responsável por encerrar função e retornar o valor informado, sendo que no comando deve ser compatível com o tipo declarado para a função.

Conhecemos o conceito de escopo de variáveis e vimos que as variáveis podem ser locais ou 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 valor do parâmetro real, pois a função trabalha com cópias dos valores passados no momento de sua chamada. Já na passagem de parâmetros por referência, esses valores são alterados, pois os parâmetros passados são endereços de memória.

Abordamos o conceito de protótipo de uma função, que informa ao compilador que outras função 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 cadeia de caracteres e construímos programas para colocar em prática os conceitos vistos.

Leitura Complementar

HERBERT, S. **C Completo e Total**. Editora Makron, 3. ed, 1997.

Sinopse: “C Completo e Total”, 3ª está dividido em cinco partes, cada uma abordando um aspecto importante de C. Parte 1 - Apresenta uma discussão detalhada da linguagem C, incluindo palavras-chaves, 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, 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.

Para saber um pouco mais sobre a construção de funções e passagem de parâmetros, acesse: <http://pt.wikibooks.org/wiki/Programar_em_C/Fun%C3%A7%C3%B5es>.

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, isto indica que um vetor ou matriz podem ser passados para uma função apenas por referência (ASCENCIO; CAMPOS, 2010).

ATIVIDADE DE AUTOESTUDO

1. Faça um programa que verifica se um determinado número é positivo ou negativo.
2. Elabore uma função que receba dois números positivos por parâmetro e retorne a soma dos N números inteiros existentes entre eles.
3. Escreva uma função que receba um caractere e retorne 1 se for uma consoante e 0 se for vogal.
4. 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. Construa uma função que receba um nome e retorne o número de vogais.
6. Elabore um programa que receba o valor da cotação do dólar, o valor em reais e apresente o valor em dólares.

7. Construa um programa que permita ao usuário gravar 10 palavras em um arquivo e, em seguida, efetue a leitura do arquivo.

EXERCICIOS DE FIXACÃO

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

Quadro 82 - Programa em C

```
#include <stdio.h>
#include <conio.h>
float convertet (float celsius);
void main ()
{
  float celsius, resposta;
  clrscr();
  printf("Informe a temperatura em graus Celsius :");
  scanf ("%f", &celsius);
  resposta = convertet(celsius);
  printf ("A temperatura %.2f em Fahreneit e %.2f", celsius, resposta);
}
float convertet (float celsius)
{
  float temp;
  temp = celsius * 1.8 + 32;
  return temp;
}
```

2. Escreva um programa utilizando função que recebe o peso de um peso em quilogramas e converte para libras.

Quadro 83 - Programa em C

```
#include <stdio.h>
#include <conio.h>
float convertep (float peso);
void main ()
{
  float peso, resposta;
  clrscr();
  printf("Informe o peso em quilogramas :");
  scanf ("%f", &peso);
  resposta = convertep(peso);
  printf ("O peso %.2f em libras e %.2f", peso, resposta);
}
float convertep (float peso)
{
  peso = peso * 2.68;
  return peso;
}
```

3. Faça uma função que receba como parâmetro um vetor com 10 números inteiros e retorne-os ordenado em forma crescente.

Quadro 84 - Programa em C

```
#include <iostream.h>
#include <conio.h>
void ordena(int v[])
{
  int i, j, aux;
  for (i=0;i<9;i++)
  {
   for (j=i+1; j<10;j++)
   {
     if (v[i] > v[j])
     {
       aux = v[i];
       v[i] = v[j];
       v[j] = aux;
     }
   }
  }
}
void main()
{
   int i, vet[10];
   clrscr();
   for (i=0;i<10;x++)
   {
     printf("Informe o elemento %d do vetor ", i);
     cin >> vet[i]
   }
   ordena(vet);
   printf ("\n Vetor ordenado");
   for (i=0;i<10;i++)
   {
     printf("%d ", vet[i]);
   }
}
```

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

Quadro 85 - Programa em C

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int conta(char nome[])
{
  int i, tam, qtd;
  tam = strlen(nome);
  nome = strupr(nome);
  qtd = 0;
  for (i=0;i<tam;i++)
  {
     if ((nome[i] != 'A') && (nome[i] !='E') && (nome[i] !='I') && (nome[i]
!='O') && (nome[i] != 'U') && (nome[i] != ' ' ))
         qtd ++;
  }
  return qtd;
}
void main()
{
   int total;
   char nome[30];
   clrscr();
   printf("Informe o nome: ");
   gets(nome);
   total = conta(nome);
   printf("\n A quantidade de consoantes do nome %s e %d", nome, total);
}
```

CONCLUSÃO

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 I vimos o histórico da Linguagem C, suas características e como o código de um programa é convertido em um arquivo executável. Conhecemos a estrutura básica de um programa em C, que é dada por:

```
#include <stdio.h>
main()
{
        conjunto de instruções;
}
```

Aprendemos que todo programa possui uma função **main**, que é chamada quando o programa é executado, como inserir comentários em nossos códigos e as regras para a nomeação dos identificadores. Conhecemos os tipos básicos de dados (**char**, **int**, **float**, **double**, **enum**, **void** e **pointer**), operadores e funções intrínsecas disponíveis na linguagem C. Vimos, também, que os tipos de dados básicos podem gerar outros tipos a partir da aplicação dos modificadores **unsigned**, **short** e **long**. Trabalhamos como realizar atribuição, entrada e saída de dados. Vimos que a atribuição é realizada pelo sinal de igualdade (=), a função para entrada de dados é o **scanf** e a saída o **printf**. Além disso, estudamos como utilizar os especificadores de formato e códigos especiais.

A Unidade II abordou a construção de programas utilizando a estrutura condicional, isto é, programas com desvio fluxo. Estudamos a estrutura condicional simples, a estrutura condicional composta e a estrutura case. Aprendemos que a estrutura condicional simples é utilizada em situações em que um conjunto de instruções deve ser executado apenas se uma condição for verdadeira. Vimos que na estrutura condicional composta é realizada a avaliação de uma única expressão lógica, no entanto temos dois caminhos para seguir, um para o resultado verdadeiro e outro para falso. A sintaxe desta estrutura é:

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

Vimos que esta estrutura case deve ser utilizada em situações mutuamente exclusivas e sua sintaxe é:

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

Colocamos em prática a construção de programas utilizando cada uma das estruturas condicionais.

Na Unidade III vimos como construir algoritmos utilizando estruturas de repetição, que permitem a execução de um trecho de código repetidas vezes. Estudamos a estrutura **for** que é utilizada quando sabemos quantas vezes o trecho de código precisa ser repetido. A sintaxe da estrutura **for** é:

```
for ( i= valor inicial; condição; incremento ou
decremento de i);
{
    <instruções>;
}
```

Vimos que 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:

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

Aprendemos que a estrutura **do while** é utilizada quando temos um número indefinido de repetições, no entanto o teste lógico é realizado no final. A sintaxe desta estrutura é:

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

A Unidade IV apresentou os conceitos relacionados a vetores, *strings*, matrizes e estruturas. Estudamos que os vetores e 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. Aprendemos como efetuar classificação e pesquisa em vetores. Estudamos como efetuar leitura e escrita de *strings* e conhecemos funções que permitem concatenar, comparar, copiar, armazenar, imprimir, converter para um valor numérico e outros. Trabalhamos a definição de novos tipos de dados utilizando estruturas (*structs*), que são capazes de armazenar informações de tipos diferentes. Estudamos como manipular essas estruturas e

como construir vetores de estruturas.

Por fim, na Unidade V, aprendemos a construir programas modulares com o uso de funções, cuja sintaxe é:

```
tipo_de_retorno nome_da_função (declaração_de_parâmetros)
{
    corpo_da_função
}
```

Estudamos o conceito de escopo de variáveis e a passagem de parâmetros por valor e por referência. Vimos que na passagem de parâmetros por valor não há alteração do valor do parâmetro real e que na passagem de parâmetros por referência esses valores são alterados. Abordamos a construção de programas utilizando protótipo de uma função, recursividade e manipulação de arquivos.

Nestas cinco unidades consolidamos o aprendizado dos conceitos inicias da Linguagem de Programação C.

Muito sucesso a você!

Professora Camila

REFERÊNCIAS

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.

GUIMARÃES, A. M.; LAGES, N. A. C. **Algoritmos e Estruturas de Dados.** Rio de Janeiro: LTC, 1994.

KERNINGHAN, B. W.; RITCHIE, D. M. **C Programming Language**. 2. ed. 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.

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**. Editora FCA, 2006.

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

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