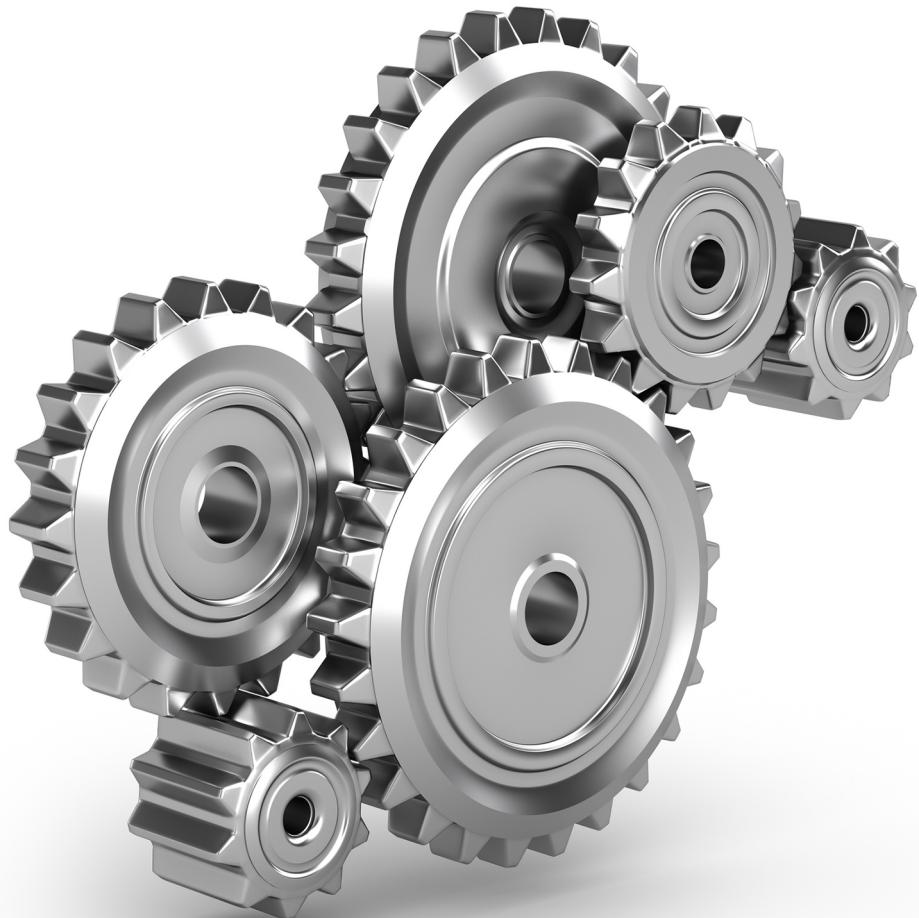


Algoritmos em C



Rodrigo L. S. Silva
Alessandreia M. Oliveira

RODRIGO LUIS DE SOUZA DA SILVA
ALESSANDREIA MARTA DE OLIVEIRA

ALGORITMOS EM C

1^a edição

Juiz de Fora
Rodrigo Luis de Souza da Silva
2014

© Rodrigo Luis de Souza da Silva e Alessandrea Marta de Oliveira

Todos os direitos estão reservados. Nenhuma parte deste livro poderá ser reproduzida ou transmitida por qualquer meio (eletrônico ou mecânico, incluindo fotocópia e gravação) ou arquivada em qualquer sistema ou banco de dados sem permissão, por escrito, dos autores.

Copyright da ilustração de capa: © *Alexandr Mitiuc* - Fotolia.com

Nota: apesar dos cuidados e revisões, erros de digitação, ortográficos e dúvidas conceituais podem ocorrer. Em qualquer hipótese, solicitamos a comunicação para o e-mail livroalgoritmos@gmail.com, para que possamos esclarecer ou encaminhar a questão.

R696a

Silva, Rodrigo Luis. Oliveira, Alessandrea Marta.
Algoritmos em C / Rodrigo Luis de Souza da Silva,
Alessandrea Marta de Oliveira. – Juiz de Fora, 2014
118 p. ; 21 cm.

ISBN: **978-85-917697-1-1**

1. Algoritmos. 2. Linguagem de Programação. I. Título.

CDD: 005.1
CDU: 004.421

Prefácio

Este material foi desenvolvido para atender às disciplinas básicas de programação do Departamento de Ciência da Computação (DCC) da Universidade Federal de Juiz de Fora (UFJF) que atende diversos cursos desta universidade. Ele abrange o conteúdo programático das disciplinas de Algoritmos e Laboratório de Programação. A linguagem utilizada neste livro é a linguagem C.

Vários exercícios contidos neste livro foram extraídos das apresentações que foram criadas para atender às demandas destas disciplinas. Estes exercícios foram revisados e atualizados com a colaboração de todos os professores e monitores que atuaram neste curso a partir de 2009.

O primeiro capítulo deste livro tem como objetivo fornecer ao leitor conceitos básicos sobre algoritmos, pseudolinguagem, lógica de programação, linguagem de programação e ambientes de desenvolvimento integrado. O capítulo seguinte apresenta os conceitos de constantes e variáveis e suas formas de utilização. Neste capítulo também são apresentados os primeiros comandos de entrada e saída. A seguir será introduzido o conceito de funções, que permitem, por exemplo, uma melhor modularização do código. Os próximos capítulos tratam das estruturas de controle básicas que são: Sequência Simples, Alternativa e Repetição. Em seguida são apresentadas as estruturas de dados homogêneas e heterogêneas. Um capítulo adicional tratando de arquivos finaliza este trabalho.

Agradecimentos

Gostaríamos de registrar nossos agradecimentos a todos os que colaboraram para o desenvolvimento deste trabalho. Em especial, agradecemos aos professores do DCC/UFJF que colaboraram intensamente para o desenvolvimento e posterior aprimoramento deste material. Agradecemos também aos monitores da disciplina de Algoritmos e bolsistas do Grupo de Educação Tutorial do DCC (GETComp), que trabalharam na elaboração e resolução de diversos exercícios contidos neste trabalho e no material disponível na página do curso de Algoritmos da UFJF. Finalmente, gostaríamos de agradecer às alunas Joviana Fernandes Marques e Julia Dias Möller, do Instituto de Artes e Design (IAD-UFJF) pela colaboração nas ilustrações introdutórias de cada capítulo.

Sumário

1	Introdução	9
1.1	Conceitos	10
1.1.1	Ação	10
1.1.2	Estado	10
1.1.3	Processo	10
1.1.4	Padrão de Comportamento	11
1.1.5	Algoritmo	11
1.2	Pseudolinguagem	12
1.3	Lógica de Programação	12
1.4	Linguagem de Programação	13
1.5	Linguagem C	13
1.6	Ambiente Integrado de Desenvolvimento	13
1.7	Exercícios	14
2	Tipos de Dados, Variáveis e Comandos de Entrada e Saída	15
2.1	Introdução	15
2.2	Tipos de Dados	16
2.3	Constantes	17
2.4	Variáveis	18
2.5	Declaração de Variáveis	18
2.6	Comandos Básicos	19
2.7	Comentários	21
2.8	Bloco	21
2.9	Comandos de Impressão e Leitura	21
2.9.1	Comandos de Impressão	21
2.9.2	Códigos de Impressão	22
2.9.3	Fixando as Casas Decimais	23
2.9.4	Alinhamento de Saída	24
2.9.5	Comandos de Leitura	25
2.10	Estruturas de Controle	25
2.11	Exercícios Resolvidos	26
2.12	Exercícios	28

3	Funções	30
3.1	Definição	30
3.2	Análise Semântica e Sintática	34
3.3	Escopo de Variáveis	34
3.4	Declaração e Definição	35
3.5	Passagem de Parâmetros	36
3.6	Recursividade	37
3.7	Exercícios Resolvidos	37
3.8	Exercícios	39
4	Estruturas Condicionais	40
4.1	Introdução	40
4.2	Alternativa Simples	41
4.3	Alternativa Dupla	42
4.4	Múltipla Escolha	44
4.5	Exercícios Resolvidos	45
4.6	Exercícios	49
5	Comandos de Repetição	51
5.1	Tipos de Repetição	51
5.1.1	Repetição com Teste no Início	52
5.1.2	Repetição com Teste no Fim	52
5.1.3	Repetição com Variável de Controle	52
5.2	Usos Comuns de Estruturas de Repetição	53
5.2.1	Repetição com <i>Flags</i>	53
5.2.2	Repetição com Acumuladores	54
5.2.3	Repetição com Contadores	56
5.3	Resumo das Estruturas de Controle	57
5.4	Exercícios Resolvidos	57
5.5	Exercícios	62
6	Vetores Numéricos	64
6.1	Introdução	64
6.2	Motivação	65
6.3	Variáveis Compostas Homogêneas	66
6.4	Vetores	66
6.5	Vetores e Funções	69
6.6	Exercícios Resolvidos	69
6.7	Exercícios	72

7	Vetores de Caracteres	73
7.1	Cadeia de Caracteres	74
7.2	<i>Strings</i>	75
7.3	Comandos de Leitura e Impressão	76
7.4	Exercícios Resolvidos	79
7.5	Exercícios	82
8	Vetores Multidimensionais	83
8.1	Definição	83
8.2	Declaração e Atribuição	84
8.3	Matrizes e Funções	87
8.4	Exercícios Resolvidos	87
8.5	Exercícios	90
9	Estruturas	91
9.1	Estruturas de Dados Heterogêneas	91
9.2	Definição	92
9.3	Manipulação	93
9.4	Vetores de Estruturas	98
9.5	Exercícios Resolvidos	98
9.6	Exercícios	103
Apêndice A	Arquivos	105
A.1	Abertura e Fechamento de Arquivos	106
A.2	Leitura e Escrita em Arquivos	109
A.3	Outros Comandos	112
Sobre os Autores		116
Referências Bibliográficas		117

1 Introdução



O uso de algoritmos surgiu como uma forma de indicar caminhos para a solução dos mais variados problemas. Dado um problema, os principais cuidados daquele que for resolvê-lo utilizando algoritmos são:

- Entender perfeitamente o problema;
- Escolher métodos para sua solução;
- Desenvolver um algoritmo baseado nos métodos;
- Codificar o algoritmo na linguagem de programação disponível.

A parte mais importante da tarefa de programar é a construção de algoritmos. Segundo Niklaus Wirth: “Programação é a arte de construir e formular algoritmos de forma sistemática”.

O aprendizado de algoritmos não se consegue a não ser através de muitos exercícios. Não se aprende Algoritmos copiando ou estudando códigos. **Algoritmos só se aprende construindo e testando códigos.**

1.1 Conceitos

Serão apresentados a seguir alguns conceitos básicos relacionados à construção de algoritmos.

1.1.1 Ação

Ação é um evento que ocorre num período de tempo finito, estabelecendo um efeito intencionado e bem definido. Como exemplos, podem ser citados: “Colocar o livro em cima da mesa”; “Atribuir o valor 3,1416 a uma variável”.

Toda ação deve ser executada em um tempo finito (do instante t_0 até o instante t_1). O que realmente interessa é o efeito produzido na execução da ação. Pode-se descrever o efeito de uma ação comparando o estado no instante t_0 com o estado no instante t_1 .

1.1.2 Estado

Estado de um dado sistema de objetos é o conjunto de propriedades desses objetos que são relevantes em uma determinada situação, como por exemplo: “Livro na estante ou sobre a mesa”; “Conjunto de valores das variáveis do programa num certo instante da execução”.

1.1.3 Processo

Processo é um evento considerado como uma sequência temporal de ações, cujo efeito total é igual ao efeito acumulado dessas ações.

Pode-se, geralmente, considerar um mesmo evento como uma ação ou como um processo, dependendo se o interesse está simplesmente no efeito total (da ação) ou em um ou mais estados intermediários (do processo). Em outras palavras, se há interesse, uma ação pode ser geralmente detalhada em um processo.

1.1.4 Padrão de Comportamento

Em todo evento pode-se reconhecer um padrão de comportamento, isto é, cada vez que o padrão de comportamento é seguido, o evento ocorre.

Como exemplo, pode-se considerar a seguinte descrição: “Uma dona de casa descasca as batatas para o jantar”; “Traz a cesta com batatas da dispensa”; “Traz a panela do armário”; “Descasca as batatas”; “Coloca a cesta a dispensa”. Isto pode ser usado para descrever eventos distintos (dias diferentes, batatas diferentes etc.) porque eles têm o mesmo padrão de comportamento.

O efeito de um evento fica totalmente determinado pelo padrão de comportamento e eventualmente pelo estado inicial.

1.1.5 Algoritmo

Algoritmo é a descrição de um padrão de comportamento, expresso em termos de um repertório bem definido e finito de ações primitivas que, com certeza, podem ser executadas. Possui caráter imperativo, razão pela qual uma ação em um algoritmo é chamada de comando.

O exemplo a seguir apresenta um algoritmo para descascar batatas para o jantar:

“**Trazer a cesta com batatas da dispensa**”;
“**Trazer a panela do armário**”;
“**Descascar as batatas**”;
“**Colocar a cesta na dispensa**”.

Um algoritmo (ou programa) apresenta dois aspectos complementares, o dinâmico e o estático. O aspecto dinâmico é a execução do algoritmo no tempo. Já o aspecto estático é a representação concreta do algoritmo, através de um texto, contendo comandos que devem ser executados numa ordem prescrita (atemporal).

O problema central da computação está em relacionar esses dois aspectos, isto é, consiste no entendimento (visualização) das estruturas dinâmicas das possíveis execuções do algoritmo a partir da estrutura estática do seu texto.

A restrição a um número limitado de estruturas de controle de execução dos comandos do algoritmo permite reduzir a distância existente entre os

aspectos estático e dinâmico. Para tal, são usadas somente três estruturas de controle: sequência simples, alternativa e repetição. Estas estruturas podem ser organizadas em uma pseudolinguagem ou pseudocódigo.

A generalização do algoritmo para descascar batatas para o jantar pode ser dada por:

“Trazer a cesta com batatas da dispensa”;
“Trazer a panela do armário”;
se “saia é clara” // alternativa
 então “Colocar avental”;
enquanto “número de batatas é insuficiente” faça // repetição
 “Descascar uma batata”;
“Colocar a cesta na dispensa”;

Um algoritmo deve ser determinístico, isto é, dadas as mesmas condições iniciais, deve produzir em sua execução os mesmos resultados. Em nosso curso, somente interessam os algoritmos executáveis em tempo finito.

Existem vários livros que tratam de Algoritmos de forma mais ou menos abrangente. Entre os livros mais comumente utilizados pode-se destacar o do Guimarães e Lages (Guimarães e Lages, 1994) e o do Cormen (Cormen, 2012).

1.2 Pseudolinguagem

Uma pseudolinguagem é uma forma genérica de escrever um algoritmo, utilizando uma forma intermediária entre a linguagem natural e uma linguagem de programação. É uma linguagem que não pode ser executada num sistema real (computador). Existem vários tipos de pseudolinguagem atualmente na literatura. Muitas delas são próximas à linguagem de programação Pascal.

1.3 Lógica de Programação

O desenvolvimento de um programa requer a utilização de um raciocínio ímpar em relação aos raciocínios utilizados na solução de problemas de outros campos do saber. Para resolver um determinado problema é necessário encontrar uma sequência de instruções cuja execução resulte na solução da questão.

Pode-se dizer que um programa é um algoritmo que pode ser executado em um computador. Já a lógica de programação é um conjunto de raciocínios utilizados para o desenvolvimento de algoritmos.

1.4 Linguagem de Programação

É um conjunto de regras sintáticas e semânticas usadas para definir um programa de computador. Uma linguagem permite que um programador especifique precisamente sobre quais dados um computador vai atuar, como estes dados serão armazenados ou transmitidos e quais ações devem ser tomadas sob várias circunstâncias.

1.5 Linguagem C

A linguagem de programação utilizada neste material é a linguagem C. As bases da linguagem C foram desenvolvidas entre 1969 e 1973, em paralelo com o desenvolvimento do sistema operacional Unix.

Esta linguagem é amplamente utilizada, principalmente no meio acadêmico. O sucesso do sistema operacional Unix auxiliou na popularização do C. Há diversos materiais *online* em português que dão suporte a esta linguagem como o material disponibilizado pela UFMG (Apostila de C - UFMG, 2005), entre outros.

Existem inúmeros livros que exploram esta linguagem tais como (Schildt, 2005; Kernighan, 1989; Deitel, 2011).

1.6 Ambiente Integrado de Desenvolvimento

O Ambiente Integrado de Desenvolvimento (ou IDE - *Integrated Development Environment*) é um programa de computador que reúne características e ferramentas de apoio ao desenvolvimento de software com o objetivo de agilizar este processo.

O *CodeBlocks* é um IDE disponível para Linux e Windows e o *download* gratuito pode ser obtido em <http://www.codeblocks.org/downloads>.

1.7 Exercícios

1. Qual o padrão de comportamento utilizado para gerar esta sequência?
1, 5, 9, 13, 17, 21, 25
2. Qual o padrão de comportamento utilizado para gerar esta sequência?
1, 1, 2, 3, 5, 8, 13, 21, 34
3. Qual o padrão de comportamento utilizado para gerar esta sequência?
1, 4, 9, 16, 25, 36, 49, 64, 81
4. Elaborar um algoritmo para descrever como você faz para ir da sua casa até o supermercado mais próximo.
5. Elaborar um algoritmo para descrever como você faz para trocar o pneu do seu carro. Considerar que você estava dirigindo quando percebeu que o pneu furou.
6. Citar 2 exemplos de IDE (*Integrated Development Environment*) que permitem o desenvolvimento de programas na linguagem C.
7. Citar 5 exemplos de linguagem de programação utilizadas atualmente.
8. Que características um bom programa precisa ter?
9. Elaborar um algoritmo para calcular a soma de um conjunto de números. Usar como base o algoritmo para descascar batatas para o jantar visto anteriormente.
10. Elaborar um algoritmo para calcular a média das duas notas de um aluno em uma disciplina. Se o aluno obtiver média maior ou igual a 70 ele será aprovado. Usar como base o algoritmo para descascar batatas para o jantar visto anteriormente.

2 Tipos de Dados, Variáveis e Comandos de Entrada e Saída



João possui vários objetos e precisa encaixá-los nos locais apropriados. De acordo com as características de cada um dos objetos, eles somente se encaixam em um determinado local. Se João tentar colocar um objeto triangular em um buraco circular, por exemplo, não vai encaixar...

Este capítulo mostrará que determinadas informações somente poderão ser armazenadas em determinados “recipientes” chamados variáveis.

2.1 Introdução

Como padrão, para cada comando serão apresentadas a sua sintaxe e a sua semântica. A sintaxe é o formato geral do comando e deve ser aceita e respeitada como padrão. A semântica, por sua vez, corresponde ao significado da ação realizada pelo comando, em tempo de execução.

2.2 Tipos de Dados

Em um algoritmo, um valor será representado na forma de constante ou de variável e terá um tipo associado. Um tipo de dados é constituído de duas partes: um conjunto de objetos (domínio de dados) e um conjunto de operações aplicáveis aos objetos do domínio.

Toda linguagem de programação possui um conjunto de tipos de dados, também chamados de implícitos, primitivos ou básicos. Os tipos de dados básicos são: inteiro, real, caractere e lógico. Em C, os tipos são, respectivamente, *int*, *float/double* e *char*. Não há o tipo lógico em C.

A ocupação em memória e a faixa de valores possíveis para esses tipos de dados em C podem ser observadas na tabela a seguir.

Tipo	Ocupação	Faixa
char	1 byte	-128 a 127 (incluindo letras e símbolos)
int	4 bytes	-2147483648 a 2147483647
float	4 bytes	3.4E-38 a 3.4E+38 (6 casas de precisão)
double	8 bytes	1.7E-308 a 1.7E+308 (15 casas de precisão)

A seguir são apresentados os tipos de dados básicos, bem como o domínio e as operações associados a cada um deles.

Inteiro

Domínio: conjunto dos inteiros.

Operações:

+, -, *, /, % (resto da divisão) geram um resultado inteiro.

<, <=, >, >=, == (igual) e != (diferente) geram um resultado lógico.

Real

Domínio: conjunto dos reais.

Operações:

+, -, * e / geram um resultado real.

<, <=, >, >=, == e != geram um resultado lógico.

Caractere

Domínio: conjunto de caracteres alfanuméricos.

Operações:

<, <=, >, >=, == e != geram um resultado lógico.

Lógico

Domínio: {verdadeiro, falso}.

Operações:

Conectivos lógicos:

Conjunção (`&&` que corresponde ao e lógico)

Disjunção (`||` que corresponde ao ou lógico)

Negação (`!` que corresponde ao não lógico).

Conectivos relacionais:

`==` e `!=` geram um resultado lógico.

Como visto na tabela, o operador “/” efetua a divisão do primeiro número inteiro pelo segundo número e tem como resultado a parte inteira desta divisão ($7 / 2 = 3$). O operador % efetua a divisão do primeiro número inteiro pelo segundo número também inteiro e tem como resultado o resto desta divisão ($7 \% 2 = 1$). Além disso, vimos também que o símbolo “==” é utilizado para efetuar comparações entre dois valores. O símbolo “=” em C é utilizado para operações de atribuição (por exemplo, em $x = a + b$, lê-se x recebe o resultado da soma de $a + b$).

2.3 Constantes

Uma constante é representada em um programa diretamente pelo seu valor que não se altera durante a execução do mesmo.

Em C, as constantes são definidas através da diretiva *define*. O tipo básico associado a uma constante fica determinado pela própria apresentação da constante. Normalmente as constantes são definidas com todas as letras maiúsculas como nos exemplos a seguir.

```
1 #define PI 3.1415
2 #define TAMANHO 10
```

2.4 Variáveis

Uma variável é representada no texto de um programa por um nome que corresponde a uma posição da memória que contém o seu valor. Em tempo de execução, o nome da variável permanecerá sempre o mesmo e seu valor pode ser modificado.

O nome de uma variável ou identificador é criado pelo programador e deve ser iniciado por uma letra que pode ser seguida por tantas letras, algarismos ou sublinhas quanto se desejar e é aconselhável que seja significativo. Como exemplos de nomes de variáveis tem-se:

Certo: nome, telefone, salario_func, x1, nota1, Media, SOMA

Errado: 1ano, sal/hora, _nome

Vale lembrar que a linguagem C é *case sensitive*, ou seja, as letras maiúsculas diferem das minúsculas.

2.5 Declaração de Variáveis

A declaração das variáveis deve ser feita no início do programa ou de um bloco com a sintaxe a seguir. Podemos criar mais de uma variável do mesmo tipo separando os identificadores por vírgula.

```
1 <tipo> <nome_var>;
```

Declarar uma variável implica em efetuar a alocação de um espaço na memória que possa conter um valor do seu tipo e uma associação do endereço dessa posição da memória ao nome da variável. Alguns exemplos de declaração de variáveis são vistos a seguir.

```
1 char f, n;
2 int idade;
3 float a, b, X1;
4 double num1, num2;
```

Toda e qualquer variável deve ser declarada e sua declaração deve ser feita antes de sua utilização no programa. Toda vez que uma variável declarada for referenciada em qualquer ponto do programa, será utilizado o conteúdo de seu endereço que corresponde ao valor da variável.

Uma variável não pode ter o mesmo nome de uma palavra-chave da linguagem C, como por exemplo: *main*, *int*, *float*, *char*, *short*, *return*, *case*, *void* etc.

As variáveis podem armazenar somente informações ou dados de um mesmo tipo (inteiro, real e caractere).

2.6 Comandos Básicos

Existem alguns comandos básicos que podem ser utilizados no desenvolvimento dos algoritmos.

Conforme visto, o operador de atribuição em C é o sinal de igual = (syntaxe: <variavel> = <expressão>). Os exemplos a seguir ilustram a utilização dos comandos de atribuição na linguagem C.

```
1 int a, b, c, d;
2 a = 5;
3 c = 7;
4 b = a;
5 d = a + b + c;
```

ou

```
1 int a = 5;
2 int c = 7;
3 int b, d;
4 b = a;
5 d = a + b + c;
```

As expressões aritméticas fornecem resultado numérico (inteiro ou real). Os operadores básicos são +, -, * e /. A seguir alguns exemplos da utilização destes operadores na linguagem C.

```
1 a = a + b;
2 a = a + 4;
3 a = b / 2;
4 a = 4 * 2 + 3;
5 b = 2 * 3 - 2 * 2;
```

A tabela a seguir apresenta os operadores aritméticos em C. Os operadores que utilizam dois argumentos são classificados como binários e os que utilizam apenas um são classificados como unários.

Aritmética	Tipo	Opção
+	Binário	Adição
-	Binário	Subtração
%	Binário	Resto da divisão
*	Binário	Multiplicação
/	Binário	Divisão
++	Unário	Incremento
--	Unário	Decremento
+	Unário	Manutenção do sinal
-	Unário	Inversão do sinal

Algumas operações matemáticas são representadas através de funções (detalhes a respeito de funções serão tratados no capítulo 3). Por exemplo, para representar a operação x^n podemos usar a função $pow(x, n)$ que retornará o resultado da expressão.

Sobre a prioridade de execução das operações em uma expressão, algumas regras devem ser seguidas:

- 1º. Parênteses (dos mais internos para os mais externos)
- 2º. Expressões aritméticas, seguindo a ordem: funções, * e /, + e -
- 3º. Conectivos relacionais: <, <=, >, >=, == e !=
- 4º. não
- 5º. e
- 6º. ou
- 7º. Da esquerda para a direita quando houver indeterminações.

Existem também as expressões lógicas que fornecem como resultado **verdadeiro** ou **falso**. Como visto anteriormente, os conectivos lógicos estão representados por conjunção (e), disjunção (ou) e negação (não). A tabela a seguir é denominada “tabela verdade” e exemplifica a utilização destes operadores, onde A e B são duas expressões lógicas.

A	B	A <u>e</u> B	A <u>ou</u> B	<u>não</u> A
V	V	V	V	F
V	F	F	V	F
F	V	F	V	V
F	F	F	F	V

2.7 Comentários

Comentários são trechos do código que não serão interpretados pelo compilador. Os comentários podem ser utilizados colocando-se duas barras “//” antes do texto. No exemplo a seguir tanto a primeira linha quanto a segunda possuem comentários. Outra possibilidade é criar comentários contendo várias linhas utilizando os caracteres “/*” no início e “*/” no final como exemplificado nas linhas 4 a 7.

```
1 int maior; // maior valor lido
2 // ler os valores das variáveis A, B, C
3
4 /*
5 Exemplo de comentário
6 em bloco
7 */
```

2.8 Bloco

Blocos são comandos delimitados por chaves ({ }). Pode-se declarar variáveis em seu interior e delimitar seu escopo como visto a seguir.

```
1 {
2     <declaração de variáveis>;
3     <comandos>;
4 }
```

2.9 Comandos de Impressão e Leitura

Comandos de impressão e leitura são utilizados para fornecer meios de interação entre o usuário e o sistema. Os comandos de impressão são utilizados para representar valores a serem informados para o usuário. Já os de leitura permitem que o usuário forneça informações ao sistema.

2.9.1 Comandos de Impressão

A exibição de mensagens em C é feita através da função predefinida *printf()*, cujo protótipo está contido no arquivo *stdio.h*. Sua sintaxe é mostrada a seguir.

```
1 printf("expressão" , lista de argumentos );
```

Neste comando, “expressão” contém mensagens a serem exibidas, códigos de formatação que indicam como o conteúdo de uma variável deve ser exibido e códigos especiais para a exibição de alguns caracteres especiais. A “lista de argumentos” pode conter identificadores de variáveis, expressões aritméticas ou lógicas e valores constantes. Veja a seguir um exemplo utilizando a função `printf` que possui apenas uma expressão.

```

1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Estou aprendendo algoritmos.");
6     return 0;
7 }
```

2.9.2 Códigos de Impressão

A tabela a seguir mostra alguns elementos extras que podem ser utilizados na impressão de tipos de dados.

Código	Tipo	Elemento armazenado
%c	char	um único caractere
%d	int	um inteiro
%f	float	um número em ponto flutuante
%lf	double	ponto flutuante com dupla precisão
%e	float ou double	um número na notação científica
%s	-	uma cadeia de caracteres

A tabela a seguir apresenta alguns códigos especiais.

Código	Ação
\n	leva o cursor para a próxima linha
\t	executa uma tabulação
\b	executa um retrocesso
\f	leva o cursor para a próxima página
\a	emite um sinal sonoro (beep)
\"	exibe o caractere ”
\\	exibe o caractere \
%%	exibe o caractere %

Alguns exemplos de utilização do comando *printf* com elementos extras e códigos especiais são apresentados a seguir.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Valor recebido foi %d", 10);
6     return 0;
7 }
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Caracter A: %c", 'A');
6     return 0;
7 }
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Valor inteiro %d e float %f",10 ,1.10);
6     return 0;
7 }
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("\t\ttx\n");
6     printf("\tx\t\ttx\n");
7     printf("\t\ttx\n");
8     return 0;
9 }
```

2.9.3 Fixando as Casas Decimais

Por padrão, a maioria dos compiladores C exibe os números em ponto flutuante com seis casas decimais. Para alterar este número pode-se acrescentar `.n` ao código de formatação da saída, sendo n o número de casas decimais pretendido, como mostra o exemplo a seguir.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Default: %f \n", 3.1415169265);
6     printf("Uma casa: %.1f \n", 3.1415169265);
7     printf("Duas casas: %.2f \n", 3.1415169265);
8     printf("Tres casas: %.3f \n", 3.1415169265);
9     printf("Notacao científica: %e \n", 3.1415169265);
10    return 0;
11 }
```

A saída para este exemplo é mostrada a seguir.

```
1 Default: 3.141517
2 Uma casa: 3.1
3 Duas casas: 3.14
4 Tres casas: 3.142
5 Notacao científica: 3.141517e+000
```

2.9.4 Alinhamento de Saída

Pode-se fixar a coluna da tela a partir da qual o conteúdo de uma variável ou o valor de uma constante será exibido. Para isto, é necessário acrescentar um inteiro m ao código de formatação. Veja a seguir um código de exemplo e sua respectiva saída.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Valor: %d \n", 25);
6     printf("Valor: %10d \n", 25);
7     return 0;
8 }
```

```
1 Valor: 25
2 Valor:      25
```

2.9.5 Comandos de Leitura

Ler dados em C significa, através da entrada padrão (teclado), informar valores na execução de um programa. O principal comando de leitura é o *scanf* e sua sintaxe é mostrada a seguir.

```
1 scanf("expressão de controle", argumentos);
```

Um exemplo de utilização do *scanf* é apresentado a seguir. O uso do & antes das variáveis *n1* e *n2* é obrigatório. Este símbolo representa o endereço de memória da variável. Este endereço deve ser informado para que a variável possa ser alterada dentro da função *scanf*.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int n1, n2, soma;
6     printf("Digite dois valores: ");
7     scanf("%d", &n1);
8     scanf("%d", &n2);
9     soma = n1 + n2;
10    printf("\n%d + %d = %d.", n1, n2, soma);
11    return 0;
12 }
```

2.10 Estruturas de Controle

Serão tratadas três estruturas de controle sendo a primeira denominada sequência simples, que como o próprio nome diz, indica uma execução sequencial dos comandos. O controle de fluxo de execução entra na estrutura, executa comando por comando, de cima para baixo e sai da estrutura. Veja a seguir um exemplo em C onde cada linha representa uma etapa da sequência.

```
1 scanf("%d %d", &x, &y);
2 a = x + y;
3 b = x - y;
4 printf ("%d %d", a, b);
```

Demais estruturas de controle serão apresentadas nos Capítulos 4 e 5.

2.11 Exercícios Resolvidos

Veremos a seguir alguns problemas envolvendo tipos de dados e funções de entrada e saída.

Problema 1

Desenvolver um algoritmo para ler um salário (real) e um percentual (inteiro) de aumento. O algoritmo deverá imprimir o salário atualizado com o aumento proposto com duas casas decimais.

Solução

Uma das possíveis soluções para este problema envolve a utilização de três variáveis: uma para receber o salário, outra para o percentual e uma última para armazenar o salário atualizado. Após a leitura do salário e do percentual, deve ser feito o cálculo correto e a atribuição do resultado para a variável correspondente. Ao final, serão utilizadas funções apropriadas para impressão como mostrado no código a seguir.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     float salario, salarioAtualizado;
6     int percentual;
7     printf("Informe o salário atual: ");
8     scanf("%f", &salario);
9     printf("Informe o percentual do aumento: ");
10    scanf("%d", &percentual);
11
12    salarioAtualizado = salario + salario * percentual / 100.0;
13
14    printf("Salário atualizado: %.2f\n", salarioAtualizado);
15    return 0;
16 }
```

Problema 2

Desenvolver um algoritmo para receber o valor de um veículo e do seu IPVA. Este algoritmo deverá imprimir quantos porcento do valor do veículo corresponde o seu IPVA no seguinte formato: “O IPVA corresponde a x% do seu valor.” (sendo ‘x’ o valor correspondente).

Solução Proposta

Após a leitura do valor do veículo e do seu IPVA deve ser feito o cálculo que nos forneça o percentual correspondente. Para a impressão deve ser usado o símbolo `%%` para que a saída seja como solicitada, como a seguir.

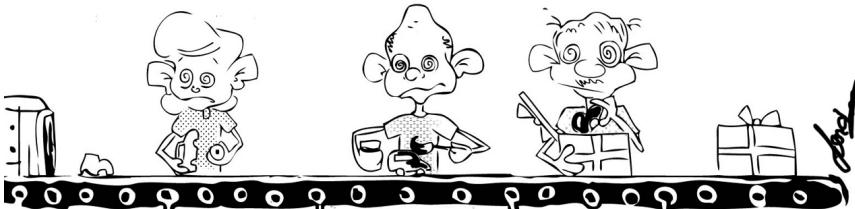
```
1 #include <stdio.h>
2
3 int main()
4 {
5     float valor, ipva, perc;
6     printf("Informe o valor do veiculo: ");
7     scanf("%f", &valor);
8     printf("Informe o ipva: ");
9     scanf("%f", &ipva);
10    perc = 100 * ipva / valor;
11
12    printf("O IPVA corresponde a %.1f%% do seu valor.\n", perc);
13    return 0;
14 }
15 }
```

2.12 Exercícios

1. Sendo $A=3$, $B=7$ e $C=4$, informar se as expressões a seguir são verdadeiras ou falsas.
 - (a) $(A + C) > B$
 - (b) $B \geq (A + 2)$
 - (c) $C = (B - A)$
 - (d) $(B + A) \leq C$
 - (e) $(C + A) > B$
2. Sendo $A=5$, $B=4$ e $C=3$ e $D=6$, informar se as expressões a seguir são verdadeiras ou falsas.
 - (a) $(A > C) \underline{\text{e}} (C \leq D)$
 - (b) $(A + B) > 10$ ou $(A + B) = (C + D)$
 - (c) $(A \geq C) \underline{\text{e}} (D \geq C)$
3. Determinar os resultados obtidos na avaliação das expressões lógicas seguintes, sabendo que A , B e C contêm respectivamente 2, 7, 3.5 e que existe uma variável lógica $L1$ cujo valor é falso.
 - (a) $B = A*C \underline{\text{e}} L1$
 - (b) $A + C < 5$
 - (c) $A*C/B > A*B*C$
 - (d) não FALSO
4. Determinar o resultado lógico das expressões mencionadas (Verdadeira ou Falsa). Considerar para as respostas os seguintes valores: $X=1$, $A=3$, $B=5$, $C=8$ e $D=7$.
 - (a) não ($X > 3$)
 - (b) $(X < 1) \underline{\text{e}} (\underline{\text{não}} (B > D))$
 - (c) não ($D < 0$) e ($C > 5$)
 - (d) não ($(X > 3)$ ou ($C < 7$))
 - (e) $(A > B)$ ou ($C > B$)
 - (f) $(X \geq 2)$
 - (g) $(X < 1) \underline{\text{e}} (B \geq D)$

- (h) $(D < 0)$ **ou** $(C > 5)$
- (i) **não** $(D > 3)$ **ou** (**não** $(B < 7)$)
- (j) $(A > B)$ **ou** (**não** $(C > B)$)
5. Fazer um programa para imprimir o seu nome.
 6. Modificar o programa anterior para imprimir na primeira linha o seu nome, na segunda linha a sua idade e na terceira sua altura.
 7. Imprimir o valor 2.346728 com 1, 2, 3 e 5 casas decimais.
 8. Ler uma temperatura em graus Celsius e apresentá-la convertida em Fahrenheit. A fórmula de conversão: $F = (9 * C + 160)/5$.
 9. Construir um algoritmo para ler 5 valores inteiros, calcular e imprimir a soma desses valores.
 10. Construir um algoritmo para ler 6 valores reais, calcular e imprimir a média desses valores.
 11. Fazer um algoritmo para gerar e imprimir o resultado de $H = 1 + 1/2 + 1/3 + 1/4 + 1/5$.
 12. Calcular e apresentar o volume de uma lata de óleo, utilizando a fórmula $volume = 3.14159 * raio * raio * altura$.
 13. Elaborar um programa para calcular e apresentar o volume de uma caixa retangular, por meio da fórmula $volume = comprimento * largura * altura$.

3 Funções



João possuía uma fábrica de carrinhos de brinquedos e montava seus carrinhos sempre sozinho. Para otimizar a produção de sua fábrica, João contratou 3 funcionários para a realização de operações específicas. O primeiro funcionário tinha a função de montar o carrinho, o segundo a função de pintar e o terceiro a de embalar. Desta forma, João conseguiu aumentar a produção da sua empresa, pois cada funcionário tinha uma função bem específica na linha de produção e exercia esta função quantas vezes fosse preciso. João podia solicitar o trabalho de montagem, pintura e empacotamento dos brinquedos sempre que necessário.

A ideia de criar funções que são utilizadas para determinadas tarefas é amplamente utilizada no desenvolvimento de algoritmos, sendo este o tema deste capítulo. Vamos aprender aqui como criar e utilizar funções em nossos algoritmos de forma a permitir uma melhor modularização e reaproveitamento dos nossos códigos.

3.1 Definição

Podemos definir função como sendo uma parcela de código computacional que executa uma tarefa bem definida, sendo que essa tarefa pode ser executada (chamada) diversas vezes num mesmo algoritmo.

Uma das razões que motivou a utilização de funções na construção de algoritmos é a necessidade de dividir um problema computacional em pequenas

partes. Essa divisão é interessante, pois muitas destas pequenas partes tendem a se repetir durante a execução de um determinado código. Alguns exemplos de tarefas que comumente se repetem são: impressão de mensagens, realização de uma operação matricial, leitura de dados etc. Podemos sintetizar a necessidade de uso de funções quando precisarmos de:

- Utilizar uma parte do código em várias partes do algoritmo;
- Disponibilizar os mesmos códigos para vários algoritmos;
- Abstrair a complexidade e facilitar o entendimento do problema;
- Facilitar a manutenção dos códigos.

A utilização de funções facilita a programação estruturada. Dadas as fases previstas nos refinamentos sucessivos, decompõe-se o algoritmo em módulos funcionais. Tais módulos podem ser organizados/codificados como funções, ou seja, o uso de funções viabiliza a modularização dos códigos.

As funções devem executar tarefas bem definidas e não funcionam sozinhas, sendo sempre chamadas por um programa principal ou por outra função. Elas permitem a criação de variáveis próprias e a manipulação de variáveis externas (devidamente parametrizadas).

Além disso, funções facilitam a legibilidade do código através da estruturação (são agrupadas fora do programa principal) e permitem a redução do número de linha através de diversas chamadas da uma mesma função.

Até agora já fizemos uso de algumas funções disponibilizadas na linguagem C como a função de escrita (comando *printf*), a função de leitura (comando *scanf*) e a função principal do código (função *main*).

Alguns autores diferenciam as funções em dois grupos - funções que retornam algum valor e funções que não retornam nada a quem as chamou. Estas últimas são chamadas de procedimentos.

Muitos autores consideram que bons códigos são compostos por diversas pequenas funções, pois cada função deve representar apenas uma funcionalidade dentro do contexto deste código.

Uma função sempre possuirá 4 partes:

- Nome;

- Corpo;
- Lista de parâmetros de entrada;
- Tipo de retorno;

Veja a seguir como essas partes serão representadas em C.

```
1 <tipo de retorno> <Nome da função> ( <Lista de parâmetros> )
2 {
3     < Corpo da função >
4 }
```

Há situações onde não é necessário passar parâmetros para uma função. Neste caso, deve-se criá-la sem conteúdo dentro do parênteses. Quando a função não possuir valor de retorno, utilizaremos a palavra reservada *void* como tipo de retorno.

Exemplo 1

Neste primeiro exemplo, a função calcula e retorna a área de um retângulo. Esta função recebe como parâmetro a base e a altura de um determinado retângulo. Veja a seguir o código desta função.

```
1 #include <stdio.h>
2
3 float calculaAreaRetangulo(float base, float altura)
4 {
5     float area;
6     area = base * altura;
7     return area;
8 }
9
10 int main()
11 {
12     float resultado;
13     resultado = calculaAreaRetangulo(5.0, 7.3);
14     printf("Área calculada: %.2f\n", resultado);
15     return 0;
16 }
```

Podemos observar no código apresentado que a função recebeu um nome sugestivo em relação à sua funcionalidade, sendo essa uma prática desejável para melhorar a legibilidade do seu código. Neste exemplo, a função recebeu dois números reais representando a base e a altura de um retângulo.

No corpo da função, criamos uma variável para receber o cálculo desta área e utilizamos a palavra reservada *return* para retornar o valor calculado.

Já na função principal, foi criada uma variável para receber o resultado calculado e passamos como parâmetro os valores 5.0 e 7.3. Ao final, imprimimos o resultado calculado.

Dependendo das necessidades do sistema a ser desenvolvido, a função apresentada poderia imprimir o resultado calculado ao invés de retorná-lo. Neste caso, usariamos a palavra reservada *void* para indicar que a função não retornaria nenhum valor. Veja a seguir como ficaria a função com esta alteração.

```
1 #include <stdio.h>
2
3 void calculaAreaRetangulo(float base, float altura)
4 {
5     float area;
6     area = base * altura;
7     printf("Área calculada: %.2f\n", area);
8 }
9
10 int main()
11 {
12     calculaAreaRetangulo(5.0, 7.3);
13     return 0;
14 }
```

Como a função não retorna nenhum parâmetro, a sua chamada na função principal ficou simplificada, pois não há a necessidade de criarmos uma variável para receber o seu resultado, como mostra a linha 12.

É importante entendermos em que ordem os comandos serão executados quando fazemos uso de funções. Observe no exemplo a seguir a ordem dos passos (de *P1* a *P7*) em que os comandos serão executados.

```
1 #include <stdio.h>
2
3 P3 void calculaAreaRetangulo(float base, float altura)
4 {
5     P4     float area;
6     P5     area = base * altura;
7     P6     printf("Área calculada: %.2f\n", area);
8 }
9
```

```
10 P1 int main()
11 {
12     P2 calculaAreaRetangulo(5.0, 7.3);
13     P7 return 0;
14 }
```

3.2 Análise Semântica e Sintática

Durante o desenvolvimento de algoritmos devemos modelar as funções levando em consideração aspectos semânticos e sintáticos. Do ponto de vista semântico, a principal preocupação é definir corretamente a funcionalidade de uma função e também definir quais são seus dados de entrada e de saída.

Uma vez tendo sido estabelecido o aspecto semântico de uma função, podemos definir os seus detalhes sintáticos. Nesta etapa, nos preocuparemos com os tipos de dados de entrada e de saída da função. Se a função não possuir entrada, deixaremos os parênteses de sua definição sem conteúdo. Já se a função não possuir saída, utilizaremos a palavra reservada *void* no início de sua definição. Nesta etapa também definiremos quais e quantas variáveis serão utilizadas e quais estruturas de controle serão necessárias.

3.3 Escopo de Variáveis

No corpo de uma função, as chaves definem seu início e seu fim. Como visto no Capítulo 1, chaves em C definem um bloco. Desta forma, todas as variáveis criadas e parâmetros passados para uma função possuem escopo limitado e são visíveis somente dentro desta função.

Toda variável criada durante a execução de um código terá seu espaço em memória correspondente reservado pelo compilador. Esse espaço se mantém reservado durante o tempo de vida da variável. Este tempo de vida é limitado e perdura enquanto a variável estiver em execução. Após a execução de uma função, todo o espaço reservado por ela é devolvido ao sistema e o programa não poderá mais acessar esses espaços.

É importante ressaltar que uma função pode ser chamada várias vezes em um código. Em cada uma dessas chamadas novos espaços são reservados e devolvidos para o sistema após a execução da função. Por ter um escopo limitado, uma função não tem acesso a variáveis de outras funções. Com isso, duas funções podem ter variáveis com o mesmo nome sem que haja qualquer tipo de conflito.

Há um tipo de variável que mesmo não sendo criada dentro de uma função pode ser utilizada em seu escopo. Essas variáveis são denominadas globais. Variáveis globais não são declaradas dentro de um bloco e por isso podem ser utilizadas por todas as funções que compõem um código. Estas variáveis possuem espaços em memória reservados durante toda a execução do programa. Contudo, segundo vários autores, o uso de variáveis globais deve ser evitado pois são variáveis de difícil controle. Estas variáveis podem ter seus valores alterados em qualquer ponto do código, sendo difícil encontrar possíveis erros relacionados a elas.

3.4 Declaração e Definição

Toda função deve ser definida ou pelo menos declarada antes de ser utilizada. Por exemplo, as funções *printf* e *scanf* são declaradas no arquivo cabeçalho *stdio.h* e por este motivo podemos utilizá-las em nossos códigos.

Observe os dois códigos a seguir. No primeiro, a função será declarada antes da função principal e definida depois. No segundo caso, o mesmo código será alterado, posicionando a função antes de sua utilização. As duas formas podem ser utilizadas para o desenvolvimento de algoritmos.

```
1 #include <stdio.h>
2
3 // declaração da função
4 void imprimeSoma(int a, int b);
5
6 int main()
7 {
8     int a, b;
9     printf("Informe dois valores: ");
10    scanf("%d %d", &a, &b);
11    imprimeSoma(a, b); // chamada da função
12    return 0;
13 }
14
15 // definição da função após sua utilização
16 void imprimeSoma(int a, int b)
17 {
18     printf("Soma: %d\n", a + b);
19 }
```

```
1 #include <stdio.h>
2
3 // definição da função antes de sua utilização
4 void imprimeSoma(int a, int b)
5 {
6     printf("Soma: %d\n", a + b);
7 }
8
9 int main()
10 {
11     int a, b;
12     printf("Informe dois valores: ");
13     scanf("%d %d", &a, &b);
14     imprimeSoma(a, b);
15     return 0;
16 }
```

3.5 Passagem de Parâmetros

É importante ressaltar que a passagem de parâmetros para uma função pode ser feita por cópia (valor) ou por referência.

Na passagem por valor, uma cópia da variável é passada para o parâmetro da função. Qualquer alteração feita neste parâmetro não reflete em alteração na variável. Todos os exemplos vistos até aqui neste capítulo fizeram uso deste tipo de passagem de parâmetro.

A chamada por referência tem como característica alterar o conteúdo da variável que é passada como parâmetro. As variáveis são passadas com o símbolo “&”. Desta forma, o endereço em memória da variável é passado, ao invés do seu conteúdo. Na função, para representar que a passagem é feita por referência, coloca-se o símbolo “*”, indicando um ponteiro para uma determinada posição da memória.

A passagem de parâmetros por referência neste livro é utilizada de forma explícita esporadicamente. Contudo, utilizaremos passagem por referência implicitamente nos capítulos que versam sobre estruturas de dados homogêneas e heterogêneas. Maiores detalhes sobre o tema podem ser encontrados, por exemplo, em (Schildt, 2005).

3.6 Recursividade

Uma função é denominada recursiva quando tem por característica chamar a si mesma. Deve-se ter cuidado ao construir funções recursivas, pois em algum momento ela deve parar seu processamento e retornar (ou imprimir) a informação desejada. A principal utilidade de funções recursivas é a de expressar algoritmos complexos de maneira mais clara e concisa. O uso de funções recursivas está fora do escopo deste livro.

3.7 Exercícios Resolvidos

A seguir alguns problemas relacionados ao uso de funções e suas soluções.

Problema 1

Desenvolver uma função que receba 4 valores inteiros e retorne a média desses valores. Esta função deve ser chamada pela função principal.

Solução

Para este problema, os parâmetros de entrada serão os 4 valores inteiros utilizados para o cálculo da média. Como o resultado pode não ser inteiro, o parâmetro de saída será um número real. Criaremos para este problema um variável auxiliar dentro da função para receber o resultado.

```
1 #include <stdio.h>
2
3 float calculaMedia(int a, int b, int c, int d)
4 {
5     float media;
6     media = (a + b + c + d) / 4.0;
7     return media;
8 }
9
10 int main(void)
11 {
12     int v1, v2, v3, v4;
13     float media;
14     printf("Informe 4 valores inteiros: ");
15     scanf("%d %d %d %d", &v1, &v2, &v3, &v4);
16     media = calculaMedia(v1, v2, v3, v4);
17     printf("Media dos valores informados: %.2f", media);
18     return 0;
19 }
```

Problema 2

Desenvolver uma função que receba uma temperatura em graus Celsius e a retorne em graus Fahrenheit. Esta função deve ser chamada pela função principal.

Solução

Considerando ser do conhecimento do programador a fórmula de conversão de graus Celsius para Fahrenheit, o problema seria definirmos os valores de entrada e de saída da função. Conforme exposto no enunciado, a função receberá um número real correspondente à temperatura em graus Celsius e retornará outro número real com essa temperatura em graus Fahrenheit.

```
1 #include <stdio.h>
2
3 float converteGraus(float celsius)
4 {
5     float fahrenheit;
6     fahrenheit = 1.8 * celsius + 32;
7     return fahrenheit;
8 }
9
10 int main(void)
11 {
12     float celsius;
13     float fahrenheit;
14     printf("Informe a temperatura em graus Celsius: ");
15     scanf("%f", &celsius);
16     fahrenheit = converteGraus(celsius);
17     printf("Temperatura em Fahrenheit: %.2f", fahrenheit);
18     return 0;
19 }
```

3.8 Exercícios

Para cada problema a seguir, faça um algoritmo em C que o solucione. Crie também uma função principal que faça o correto uso da função criada. Posteriormente, teste este algoritmo em um IDE de sua preferência (uma sugestão de IDE foi apresentada na seção 1.6).

1. Escrever uma função para receber a idade de uma pessoa em dias e imprimir essa idade expressa em anos, meses e dias.
2. Escrever uma função para receber a idade de uma pessoa em anos, meses e dias e retornar essa idade expressa em dias.
3. Escrever uma função para receber por parâmetro o tempo de duração de um experimento expresso em segundos e imprimir na tela esse mesmo tempo em horas, minutos e segundos.
4. Escrever uma função para receber por parâmetro o raio de uma esfera e calcular o seu volume: $V = (4 * PI * R^3)/3$.
5. Escrever um procedimento para receber dois números inteiros e imprimir o produto desses valores.
6. Escrever um procedimento para receber as três notas de um aluno e imprimir a média ponderada. Considerar como peso das notas os seguintes valores: 2, 3, 5.
7. Escrever um procedimento para receber os lados de um triângulo retângulo, calcular e imprimir a sua área.
8. Em uma indústria metalúrgica o custo de produção de uma peça automotiva corresponde a um custo fixo mensal de R\$5.000,00 acrescido de um custo variável de R\$55,00 por unidade produzida mais 25% de impostos sobre o custo variável. Considerar que o preço de venda dessa peça é de R\$102,00, escrever:
 - a) a função para calcular o custo da produção de x peças.
 - b) a função para retornar a receita referente a venda de x peças.
 - c) a função para calcular o lucro na venda de x peças.
9. O IMC (índice de massa corpórea) determina se uma pessoa adulta é considerada gorda, obesa, normal ou está abaixo do peso, relacionando a massa da pessoa em quilogramas com o quadrado da medida da altura em metros. Escrever uma função para receber o peso e a altura de um adulto e calcular o seu IMC.

4 Estruturas Condicionais



João comprou um carro novo e ao abastecer ficou na dúvida sobre qual combustível colocar. Havia uma placa indicando que o álcool custava 75% do valor da gasolina. João sabe que o álcool é um combustível mais interessante se estiver no máximo a 70% do valor da gasolina. Resolvido o dilema, João encheu o seu tanque com gasolina sem ter maiores dúvidas. Da mesma forma que o caso ilustrado, muitas vezes é preciso tomar uma decisão com base em uma condição ao desenvolver um algoritmo. Este capítulo trata exatamente desta questão. Serão apresentadas as estruturas condicionais ou alternativas que correspondem ao segundo tipo de estrutura de controle. Os comandos condicionais têm a capacidade de resolver problemas de decisão como o ilustrado aqui.

4.1 Introdução

A alternativa é utilizada quando a execução de uma ação depende de uma inspeção ou teste de uma condição (expressão lógica) e apresenta a sintaxe a seguir.

```
1 if (<condição>)
2 {
3     <sequência de comandos>
4 }
```

A expressão sempre será avaliada logicamente (verdadeiro ou falso). Em C, a expressão será considerada VERDADEIRA quando o seu resultado for diferente de zero e FALSA quando a expressão for igual a zero. Desta forma, sempre que um teste lógico é executado, ele retornará um valor diferente de zero quando for verdadeiro.

Existem 3 tipos de alternativas que são descritas a seguir: **Simples**, **Dupla** e **Múltipla Escolha**.

4.2 Alternativa Simples

O comando *if* (“se” em português) pode decidir se uma sequência de comandos será ou não executada. Veja a seguir sua sintaxe.

```
1 if ( <condição> )
2 {
3     <sequência de comandos>;
4 }
```

Se houver somente um comando (uma única linha) em uma determinada estrutura, não será necessário o uso de chaves.

O exemplo a seguir demonstra o uso da estrutura de alternativa simples. Neste exemplo, é criada uma função que recebe dois valores inteiros e retorna o maior valor.

```
1 #include <stdio.h>
2
3 int encontraMaior(int a, int b)
4 {
5     int maior;
6     maior = a; // será necessário apenas um teste
7     if(b > a)
8     {
9         maior = b;
10    }
11    return maior;
12 }
```

```
13 int main()
14 {
15     int a, b, maior;
16     printf("Informe dois valores: ");
17     scanf("%d %d", &a, &b);
18     maior = encontraMaior(a, b);
19     printf("\nMaior = %d", maior);
20     return 0;
21 }
```

4.3 Alternativa Dupla

O comando *if..else* (“se..senão” em português) pode decidir entre duas sequências de comandos. O fluxo de execução do algoritmo seguirá para um dos dois possíveis caminhos (sequência de comandos 1 ou 2), dependendo se a condição for verdadeira ou falsa.

```
1 if( <condição> )
2 {
3     <sequência de comandos 1>;
4 }
5 else
6 {
7     <sequência de comandos 2>;
8 }
```

No exemplo a seguir, o algoritmo imprimirá o maior valor entre *a* e *b*, utilizando a estrutura de alternativa dupla.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int a, b, maior;
6     a = 9;
7     b = 2;
8     if(a > b)
9     {
10         maior = a;
11     }
12     else
13     {
14         maior = b;
15     }
16     printf("\nMAIOR = %d", maior);
17     return 0;
18 }
```

No próximo exemplo serão lidas as variáveis *x* e *y* e seus valores impressos em ordem crescente.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     float x, y, aux;
6     printf("Digite os dois numeros: ");
7     scanf("%f %f", &x, &y);
8     printf("Conteudos originais de x e y: %f , %f \n: ", x, y);
9     if(y < x)
10    {
11        aux = x;
12        x = y;
13        y = aux;
14    }
15    printf("Conteudos de x e y ordenados: %f , %f: \n", x, y);
16    return 0;
17 }
```

O exemplo a seguir define uma função para checar a paridade de um número.

```
1 #include <stdio.h>
2
3 void checaParidade(int valor)
4 {
5     if(valor % 2 == 0)
6     {
7         printf("Numero par.");
8     }
9     else
10    {
11        printf("Numero impar.");
12    }
13 }
14
15 int main()
16 {
17     int x;
18     printf("Digite o numero:");
19     scanf("%d", &x);
20     checaParidade(x);
21     return 0;
22 }
```

4.4 Múltipla Escolha

A alternativa em múltipla escolha é utilizada quando uma variável pode assumir diferentes valores que se deseja avaliar. Estes valores podem ser do tipo inteiro ou caractere. Veja a seguir a sintaxe desta estrutura.

```
1  switch(<condição>)
2  {
3      case V1:
4          <sequência de comandos>;
5      break;
6      case V2:
7          <sequência de comandos>;
8      break;
9      . . .
10     case Vn:
11         <sequência de comandos>;
12     break;
13     default:
14         <sequência de comandos>;
15 }
```

A seguir é apresentado um exemplo de múltipla escolha para determinar a partir de um valor inteiro lido variando de 1 a 4 se estamos no verão, outono, inverno ou primavera, nesta ordem.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int epoca;
6     scanf("%d", &epoca);
7
8     switch(epoca)
9     {
10         case 1:
11             printf("Verao.");
12         break;
13         case 2:
14             printf("Outono.");
15         break;
16         case 3:
17             printf("Inverno.");
18         break;
19         case 4:
20             printf("Primavera.");
21         break;
22 }
```

```
22     default:
23         printf("Invalido.");
24     }
25     return 0;
26 }
```

4.5 Exercícios Resolvidos

Veremos a seguir alguns problemas relacionados a utilização de estruturas condicionais com suas respectivas soluções.

Problema 1

Desenvolver uma função que receba o valor total de um pedido e o número de produtos adquiridos e retorne o valor a ser pago considerando as seguintes condições:

- Pedidos acima de R\$ 100,00 com um único produto recebem desconto de 7% e frete grátis;
- Pedidos acima de R\$ 100,00 com mais de um produto, recebem desconto de 7% e pagam frete de R\$ 5,00;
- Pedidos que custam entre R\$ 70,00 e R\$ 100,00 (inclusive) recebem desconto de 6% e pagam frete R\$ 10,00;
- Demais situações não recebem desconto e pagam frete de R\$ 10,00.

Fazer uma função principal onde serão lidos o valor total do pedido e o número de produtos adquiridos em uma compra. Imprimir na função principal o valor a ser pago com duas casas decimais.

Solução

Neste problema chamaremos a função que calculará o valor total a ser pago passando como parâmetro o valor total do pedido e número de produtos comprados. Considerando as condições especificadas no enunciado, utilizaremos a estrutura de alternativa dupla para fazer o cálculo, retornando ao final o valor calculado.

```
1 #include <stdio.h>
2
3 float calculaValorPago(float valorPedido, int numeroProdutos)
4 {
5     float resultado;
6
7     if(valorPedido > 100)
8     {
9         if(numeroProdutos == 1)
10        {
11            resultado = valorPedido * 0.93;
12        }
13        else
14        {
15            resultado = valorPedido * 0.93 + 5.0;
16        }
17    }
18    else
19    {
20        if(valorPedido >= 70 && valorPedido <= 100)
21        {
22            resultado = valorPedido * 0.94 + 10.0;
23        }
24        else
25        {
26            resultado = valorPedido + 10.0;
27        }
28    }
29    return resultado;
30 }
31
32 int main()
33 {
34     float valorPedido, valorPago;
35     int numeroProdutos;
36
37     printf("Informe o valor do pedido: ");
38     scanf("%f", &valorPedido);
39     printf("Informe o numero de produtos: ");
40     scanf("%d", &numeroProdutos);
41
42     valorPago = calculaValorPago(valorPedido, numeroProdutos);
43
44     printf("Valor a ser pago: %.2f", valorPago);
45     return 0;
46 }
```

Problema 2

Desenvolver uma função que receba um número inteiro, um símbolo de operação aritmética (+, -, * e /) e outro número inteiro (nessa ordem) e imprima o resultado da operação. Tratar na função se houver divisão por zero e neste caso exibir a mensagem “Erro - Divisao por zero”. Se a operação aritmética não for uma das quatro descritas, exibir a mensagem “Erro - Operador invalido”. Criar uma função principal que faça a leitura dos números e da operação aritmética, lidos em uma única linha (Por exemplo: 2 + 3). Chame a função criada passando os valores lidos como parâmetro.

Solução

Uma das possibilidades de resolver este problema é utilizando a estrutura de múltipla escolha. Neste caso, utilizaremos como condição do comando *switch* o operador aritmético passado como parâmetro como caractere.

```
1 #include <stdio.h>
2
3 void processaSimbolo(int n1, char op, int n2)
4 {
5     switch(op)
6     {
7         case '+':
8             printf("Resultado: %d", n1+n2);
9             break;
10        case '-':
11            printf("Resultado: %d", n1-n2);
12            break;
13        case '*':
14            printf("Resultado: %d", n1*n2);
15            break;
16        case '/':
17            if(n2==0)
18            {
19                printf("Erro - Divisao por zero.\n");
20            }
21            else
22            {
23                printf("Resultado: %d", n1/n2);
24            }
25            break;
26        default:
27            printf("Erro - Operador invalido");
28    }
29 }
```

```
30 | int main()
31 | {
32 |     int num1, num2;
33 |     char operacao;
34 |
35 |     printf("Informe a operacao: ");
36 |     scanf("%d %c %d", &num1, &operacao, &num2);
37 |
38 |     processaSimbolo(num1, operacao, num2);
39 |     return 0;
40 | }
```

4.6 Exercícios

Para cada problema a seguir, escrever um algoritmo em C que o solucione. Se for pedido para desenvolver uma função para resolver o problema, crie também uma função principal que faça uso desta função.

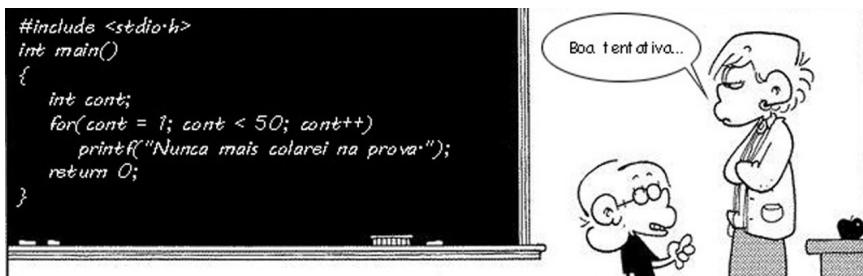
1. Ler dois números inteiros e informar se o primeiro é maior, menor ou igual ao segundo.
2. Fazer um programa para ler dois números inteiros e fazer a divisão do primeiro pelo segundo (somente se o segundo for diferente de zero).
3. Fazer uma função que receba como parâmetro um número inteiro e informe se ele é divisível por 2.
4. Alterar o algoritmo anterior para que seja informado se o número é divisível por 2 e por 3 simultaneamente.
5. Alterar o algoritmo anterior para que seja informado se o número é divisível por 2 e por 3, mas que não seja divisível por 5.
6. Fazer uma função para receber dois números reais e verificar se ambos são maiores que zero. Caso positivo, informar “Valores são válidos”. Caso contrário, informar “Valores inválidos”.
7. Tendo como dados de entrada a altura e o sexo de uma pessoa, fazer uma função para calcular e retornar seu peso ideal, utilizando as seguintes fórmulas:
para homens: $(72.7 * h) - 58$
para mulheres: $(62.1 * h) - 44.7$
8. Fazer uma função para receber três comprimentos (x , y e z) e responder se eles formam um triângulo, ou seja, se $x < y + z$ e $y < x + z$ e $z < x + y$.
9. Escrever uma função para receber a média final de um aluno por parâmetro e retornar o seu conceito, conforme a tabela a seguir.

Nota	Conceito
De 0 a 49	D
De 50 a 69	C
De 70 a 89	B
De 90 a 100	A

10. Ler o número do dia da semana e imprimir o seu respectivo nome por extenso. Considerar o número 1 como domingo, 2 para segunda etc. Caso o dia não exista (menor que 1 ou maior que 7), exibir a mensagem “Dia da semana inválido”.
11. Os funcionários de uma empresa receberam um aumento de salário: técnicos (código = 1), 50%; gerentes (código = 2), 30%; demais funcionários (código = 3), 20%. Escrever um algoritmo para ler o código do cargo de um funcionário e o valor do seu salário atual, calcular e imprimir o novo salário após o aumento.
12. Ler o valor inteiro da idade de uma pessoa e imprimir uma das mensagens dependendo das condições a seguir:
 $idade < 13$ - Criança
 $13 \leq idade < 20$ - Adolescente
 $20 \leq idade < 60$ - Adulto
 $idade \geq 60$ - Idoso
13. Ler um caractere e imprimir uma das seguintes mensagens, segundo o caso:
“Sinal de menor”
“Sinal de maior”
“Sinal de igual”
“Outro caracter”
14. Escrever uma função para retornar a divisão inteira (sem casas decimais) do dividendo pelo divisor e armazenar no parâmetro r , passado por referência, o resto da divisão. Utilizar a sugestão de protótipo a seguir.

```
int divisao (int dividendo, int divisor, int *r)  
Exemplo:  
int r, d;  
d = divisao(5, 2, &r);  
printf("Resultado: %d Resto: %d", d, r);  
// Resultado: 2 Resto: 1
```

5 Comandos de Repetição



Maria chegou para fazer sua prova de cálculo mas infelizmente não havia estudado nada. Sua melhor amiga sabia bem a matéria e Maria optou por tentar colar dela as primeiras questões. Contudo, a professora viu a tentativa de Maria e a repreendeu. Como castigo, Maria teria que escrever 49 vezes no quadro a frase “Nunca mais colarei na prova”. Maria não sabia cálculo, mas sabia algumas estruturas interessantes de Algoritmos. Uma destas estruturas é chamada de repetição que, como o nome sugere, auxiliaria Maria a repetir uma mesma ação por um número determinado de vezes. Maria então foi até o quadro e escreveu seu código para cumprir seu castigo. Infelizmente a professora não conhecia algoritmos e não aceitou a solução proposta por Maria, mas a sua ideia foi no mínimo inteligente.

Nesta aula aprenderemos como criar estruturas como a utilizada por Maria neste exemplo. As estruturas de repetição, que são o terceiro tipo de estrutura de controle que veremos neste livro, permitem que um conjunto de instruções seja repetido até que um critério de parada ocorra. Veremos neste capítulo como estas estruturas funcionam e vários exemplos de sua utilização.

5.1 Tipos de Repetição

São chamados de estruturas iterativas, iterações, laços ou *loops*. Permitem repetir a execução de uma ação várias vezes. Podem ser:

- Repetição com teste no início
- Repetição com teste no fim
- Repetição com variável de controle

5.1.1 Repetição com Teste no Início

O comando *while* (“enquanto” em português) é utilizado quando desejamos realizar uma repetição com teste no início do bloco. Este comando tem a sintaxe a seguir.

```
1 while( <condição> )
2 {
3     <sequência de comandos>;
4 }
```

Enquanto a condição for verdadeira, a sequência será repetida. Quando a condição fornecer resultado falso, o controle sai da estrutura passando para o comando seguinte ao final do bloco.

5.1.2 Repetição com Teste no Fim

O comando *do..while* (“faça..enquanto” em português) é utilizado quando desejamos realizar uma repetição com teste no final do bloco. Este comando tem a sintaxe a seguir.

```
1 do
2 {
3     <sequência de comandos>;
4 }while( <condição> );
```

Nesta estrutura, enquanto a condição for verdadeira, a sequência será repetida. Quando a condição fornecer resultado falso, o controle sai da estrutura passando para o comando seguinte. Esta estrutura se diferencia da anterior por executar a sequência de comandos pelo menos uma vez, já que o teste é feito no final.

5.1.3 Repetição com Variável de Controle

O comando *for* (“para” em português) é utilizado quando desejamos realizar uma repetição com uma ou mais variáveis de controle, sendo a inicialização e o incremento realizados dentro do próprio comando. Este comando tem a sintaxe a seguir.

```
1 for(V = I; V <= L; V = V + P)
2 {
3     <sequência de comandos>;
4 }
```

Onde V é a variável de controle, I é o valor inicial de V , L é o valor limite de V e P é o incremento sofrido por V após cada execução da sequência de comandos do bloco.

O controle do fluxo de execução entra na estrutura e faz a etapa de inicialização uma única vez ($V = I$), iniciando a estrutura de repetição na seguinte sequência:

1. executa o teste ($V \leq L$). Se for válido, avança para o passo 2. Senão, executa o primeiro comando após a estrutura de repetição;
2. executa a sequência de comandos;
3. executa o incremento/decremento ($V = V + P$);
4. retorna ao passo 1.

5.2 Usos Comuns de Estruturas de Repetição

Existem várias formas de utilização das estruturas de repetição. É possível criar estruturas de repetição que utilizem o conceito de *flags*, contadores e acumuladores. Veremos a seguir alguns exemplos.

5.2.1 Repetição com *Flags*

Flag é um valor específico escolhido pelo programador para indicar o fim dos dados de entrada. A leitura do *flag* informa ao programa que os dados de entrada terminaram. O controle de processamento genérico para uso de *flag* é mostrado a seguir.

```
1 ...
2 "leitura do valor inicial"
3 enquanto("valor não for flag")
4 {
5     "processar";
6     "leitura do próximo valor"
7 }
8 ...
```

Exemplo

Desenvolver um algoritmo para ler uma sequência de números inteiros, calcular e imprimir o quadrado de cada número lido. O último valor a ser lido é um *flag* = 0.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int num, quadrado;
6     printf("Digite um numero: ");
7     scanf("%d", &num);
8     while(num != 0)
9     {
10         quadrado = num * num;
11         printf("\nQuadrado de %d = %d: ", num, quadrado);
12         printf("Digite o proximo numero: ");
13         scanf("%d", &num);
14     }
15     return 0;
16 }
```

Teste do Algoritmo

Considerando como dados de entrada os valores 2, -3, 1, 4 e 0, veja a seguir os resultados.

num	quadrado	saída
2	4	4
-3	9	9
1	1	1
4	16	16
0		

5.2.2 Repetição com Acumuladores

Repetição com uso de acumuladores pressupõe a utilização de uma variável que acumulará um determinado valor durante a execução do algoritmo. Por exemplo, para se calcular a soma de um número indeterminado de valores, faríamos uso de uma variável que armazenaria a soma de todos os valores lidos. Esta variável é chamada acumuladora.

Exemplo

Desenvolver um algoritmo para ler uma sequência de números inteiros com *flag* = 0, calcular e imprimir a soma desses números.

Como fazer?

Uma possível solução para o somatório de valores é usar uma variável que armazenará as somas parciais. Essa variável deve ser inicializada com zero e acumular a soma de todos os números até o final da sequência.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int num, soma;
6     printf("Digite um numero: ");
7     scanf("%d", &num);
8     soma = 0; // é obrigatório inicializar com 0
9     while(num != 0)
10    {
11        soma = soma + num; // acumula os valores em SOMA
12        printf("Digite o proximo numero: ");
13        scanf("%d", &num);
14    }
15    printf("A soma dos numeros digitados foi %d.", soma);
16    return 0;
17 }
```

Teste do Algoritmo

Considerando como dados de entrada os valores 8, -3, 2, 1 e 0, veja a seguir os resultados.

num	soma	saída
	0	
8	8	
-3	5	
2	7	
1	8	
0		8

5.2.3 Repetição com Contadores

Outra função importante das estruturas de repetição é contar. Saber a quantidade de valores lidos dentro de uma repetição é importante em diferentes contextos. Para isso, utilizaremos um contador que nada mais é que um acumulador que é incrementado de 1 em 1.

Exemplo

Desenvolver um algoritmo para ler uma sequência de números inteiros com *flag* = 0, calcular e imprimir a quantidade de números lidos.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int num, cont; // 'cont' representa o contador
6     // imprime uma mensagem e lê o primeiro número inteiro
7     printf("Digite um numero inteiro: ");
8     scanf("%d", &num);
9     cont = 0;
10    while(num != 0)
11    {
12        cont = cont + 1;
13        printf("Digite o proximo numero: ");
14        scanf("%d", &num);
15    }
16    printf("Foram lidos %d numeros.", cont);
17    return 0;
18 }
```

Teste do Algoritmo

Considerando como dados de entrada os valores 8, -3, 2, 1 e 0, veja a seguir os resultados.

num	cont	saída
	0	
8	1	
-3	2	
2	3	
1	4	
0		4

5.3 Resumo das Estruturas de Controle

Uma estrutura de controle é responsável pelo fluxo de execução dos comandos que constituem o seu domínio (ou bloco). Podem ser:

1. Sequência Simples.
2. Alternativa:
 - (a) Simples (*if*).
 - (b) Dupla (*if-else*).
 - (c) Múltipla Escolha (*switch*).
3. Repetição:
 - (a) Com Teste no Início (*while*).
 - (b) Com Teste no Final (*do-while*).
 - (c) Com Variável de Controle(*for*).

5.4 Exercícios Resolvidos

Veremos a seguir alguns problemas relacionados ao uso de estruturas com repetição com suas respectivas soluções.

Problema 1

Desenvolver um algoritmo para ler uma sequência de números inteiros com *flag* = 0, calcular e imprimir a média aritmética dos números lidos.

Solução

A solução deste problema é somar todos os valores informados pelo usuário e dividir esta soma pela quantidade de números lidos.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int num;
6     int soma = 0; // "soma" representa o acumulador
7     int cont = 0; // "cont" representa o contador
8     float media;
```

```

9   printf("Digite um numero: ");
10  scanf("%d", &num);
11  while(num != 0)
12  {
13      cont = cont + 1;    // contador
14      soma = soma + num; // acumulador
15      printf("Digite o proximo numero: ");
16      scanf("%d", &num);
17  }
18  media = soma / (float) cont;
19  printf("A media dos numeros lidos foi %.2f.", media);
20  return 0;
21 }
```

Teste do Algoritmo

Considerando que serão fornecidos os seguintes dados de entrada 2, 9, 4, 22, 1, 7 e 0, veja a seguir os resultados.

num	soma	cont	media	imprime
	0	0		
2	2	1		
9	11	2		
4	15	3		
22	37	4		
1	38	5		
7	45	6		
0			7.5	7.5

Problema 2

Desenvolver uma função que receba um número inteiro positivo, calcule e retorne a soma de seus algarismos.

Solução

Por exemplo, para o número 123, a saída deve ser: $1 + 2 + 3 = 6$. Mas como separar os dígitos? Uma possibilidade seria através do uso de operadores de divisão e resto como mostrado a seguir.

$123 \% 10 = 3$ (retorna o resto da divisão de 123 por 10)
 $123 / 10 = 12$ (retorna o quociente da divisão de 123 por 10)
 $12 \% 10 = 2$ e $12 / 10 = 1$
 $1 \% 10 = 1$ e $1 / 10 = 0$

Através de sucessivas divisões por 10, encontram-se os algarismos desejados.
 $3 + 2 + 1 = 6$

```
1 #include <stdio.h>
2
3 int somaDigitos(int num)
4 {
5     int digito; // "digito" é o dígito mais a direita de 'num'
6     int soma;
7     soma = 0;
8     while(num != 0)
9     {
10         digito = num % 10; // obtém o último dígito
11         num = num / 10;    // elimina o ultimo dígito
12         soma = soma + digito;
13     }
14     return soma;
15 }
16
17 int main()
18 {
19     int num, soma;
20     printf("Digite um numero inteiro: ");
21     scanf("%d", &num);
22     soma = somaDigitos(num);
23     printf("A soma dos digitos foi %d.", soma);
24     return 0;
25 }
```

Teste do Algoritmo

Considerando o valor 374 como entrada do algoritmo, veja a seguir os resultados.

num	digito	soma	imprime
		0	
374	4	4	
37	7	11	
3	3	14	
0			14

Problema 3

Desenvolver um algoritmo para ler 100 números inteiros, calcular e imprimir o quadrado de cada número lido.

Solução

A solução consiste em fazer a leitura de cada um dos 100 números lidos e calcular o seu quadrado. Da mesma forma, a impressão deve ser feita nesta mesma estrutura de repetição.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int num, cont;
6     cont = 0;
7     do
8     {
9         printf("Digite um número: ");
10        scanf("%d", &num);
11        printf("\n%d ao quadrado = %d", num, num * num);
12        cont = cont + 1;
13    }while(cont < 100);
14    return 0;
15 }
```

Problema 4

Desenvolver uma função para imprimir os valores inteiros do intervalo fechado compreendido entre um valor inicial e um valor final passados como parâmetro. Considere que o valor inicial é sempre menor que o valor final.

Solução

Uma das possíveis soluções para este exercício consiste na criação de uma variável auxiliar que será inicializada com o menor valor e incrementada em uma unidade até que seja igual ao valor final.

```
1 #include <stdio.h>
2
3 void imprimeIntervaloFechado(int a, int b)
4 {
5     int i;
6     i = a;
7     do
8     {
9         printf("%d ", i);
10        i++; // ou i = i + 1;
11    }while(i <= b);
12 }
```

```
13 | 
14 | int main()
15 | {
16 |     int numInicial, numFinal;
17 |     printf("Digite o numero inicial: ");
18 |     scanf("%d", &numInicial);
19 |     printf("Digite o numero final: ");
20 |     scanf("%d", &numFinal);
21 |
22 |     imprimeIntervaloFechado(numInicial, numFinal);
23 |
24 |     return 0;
25 | }
```

O exercício anterior poderia ser resolvido com outras estruturas de repetição como demonstrado no exemplo a seguir.

```
1  #include <stdio.h>
2
3  void imprimeIntervaloFechado(int a, int b)
4  {
5      int i;
6      for(i = a; i <= b; i++)
7      {
8          printf("%d ", i);
9      }
10 }
11
12 int main()
13 {
14     int numInicial, numFinal;
15     printf("Digite o numero inicial: ");
16     scanf("%d", &numInicial);
17     printf("Digite o numero final: ");
18     scanf("%d", &numFinal);
19
20     imprimeIntervaloFechado(numInicial, numFinal);
21
22     return 0;
23 }
```

5.5 Exercícios

Para cada problema a seguir, faça um algoritmo em C que o solucione. Se for pedido para desenvolver uma função para resolver o problema, crie também uma função principal que faça uso desta função.

1. Desenvolver um algoritmo para imprimir todos os números pares no intervalo 1-100.
2. Desenvolver um algoritmo para imprimir todos os números de 100 até 1.
3. Escrever uma função para receber um número inteiro e positivo, verificar e informar se este é ou não um número primo.
4. Escrever uma função para receber como parâmetro dois valores inteiros $n1$ e $n2$ e imprimir o intervalo fechado entre eles, do menor para o maior.

Por exemplo: se $n1 = 2$ e $n2 = 5$, a função imprimirá 2, 3, 4, 5.

5. Escrever uma função para retornar o número de inteiros ímpares que existem entre $n1$ e $n2$ (inclusive ambos, se for o caso). A função funcionará inclusive se o valor de $n2$ for menor que $n1$.

```
n=contaimpar(10,17); // n recebe 5 (11,13,15,17)  
n=contaimpar(5,1); // n recebe 3 (1,3,5)
```

6. Escrever uma função `int somaintervalo(int n1, int n2)` para retornar a soma dos números inteiros que existem no intervalo fechado entre $n1$ e $n2$. A função funcionará inclusive se o valor de $n2$ for menor que $n1$.

```
n=somaintervalo(3, 6); // n recebe 18 (3+4+5+6)  
n=somaintervalo(5, 5); // n recebe 5 (5)  
n=somaintervalo(4, 0); // n recebe 10 (4+3+2+1+0)
```

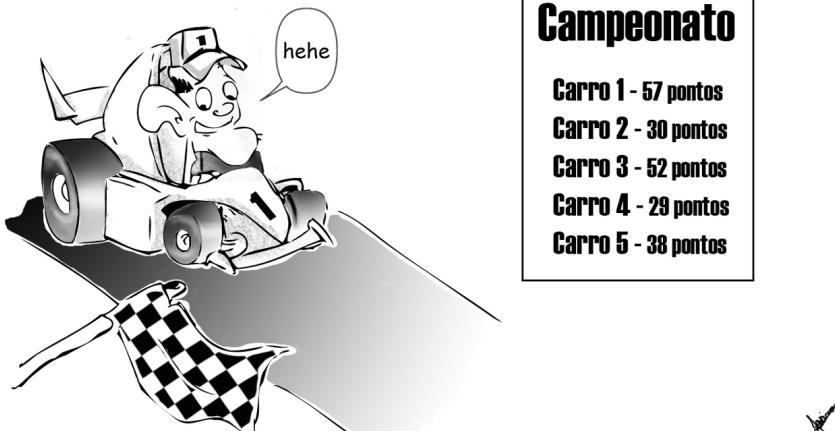
7. Dada uma dívida de R\$ 10.000,00 reais que cresce a juros de 2,5% ao mês e uma aplicação de R\$ 1.500,00 reais com rendimento de 4% ao mês, escrever um algoritmo para determinar o número de meses necessários para pagar a dívida utilizando esta aplicação.

8. Escrever uma função para receber como parâmetro um valor n inteiro e positivo e calcular o valor de S . A função retornará o valor de S .

$$S = 1 + 1/2 + 1/3 + 1/4 + \dots + 1/n$$

9. Chico tem 1,50 metro e cresce 2 centímetros por ano, enquanto Zé tem 1,40 metro e cresce 3 centímetros por ano. Construir um algoritmo para calcular e imprimir quantos anos serão necessários para que Zé seja maior que Chico.
10. Escrever um algoritmo para ler a matrícula de um aluno e suas três notas e calcular a média ponderada do aluno, considerando que o peso para a maior nota seja 4 e para as duas restantes, 3. Mostrar ao final a média calculada e uma mensagem “APROVADO” se a média for maior ou igual a 60 e “REPROVADO” caso contrário. Repetir a operação até que o código lido seja negativo.
11. Desenvolver um algoritmo que possibilite, dado um conjunto de valores inteiros e positivos (fornecidos um a um pelo usuário), determinar qual o menor valor deste conjunto. O final do conjunto de valores é conhecido através do valor zero, que não deve ser considerado.
12. A conversão de graus Fahrenheit para Celsius é obtida pela fórmula $C = (F - 32)/1.8$. Desenvolver um algoritmo para calcular e imprimir uma tabela de graus Celsius em função de graus Fahrenheit que variem de 50 a 150 de 1 em 1.
13. Elaborar um algoritmo para calcular $N!$ (fatorial de N). O valor inteiro N será fornecido pelo usuário. Considerar, por definição, que $N! = N * (N-1) * (N-2) * \dots * 3 * 2 * 1$ e $0! = 1$.
14. Fazer um algoritmo para calcular e imprimir os N primeiros termos da série de Fibonacci. O valor inteiro N será fornecido pelo usuário. Na série de Fibonacci o primeiro e o segundo termos valem 1 e os seguintes são calculados somando os dois termos anteriores.

6 Vetores Numéricos



Maria precisa armazenar os dados de uma corrida de carros que está sendo organizada em sua cidade. Para isso, ela precisa criar uma tabela contendo o número dos carros e a quantidade de pontos que cada um dos carros vai acumulando com as corridas. Maria teve a ideia de fazer um pequeno sistema para controlar esta tabela. Maria poderia criar uma variável para cada carro e ir somando a quantidade de pontos de cada carro por variável, mas o controle destas variáveis seria complicado. Maria então fez uso de uma nova estrutura chamada vetor. Nesta estrutura, Maria poderia guardar os pontos de todos os carros usando apenas uma única variável com vários índices. Neste capítulo veremos como criar e em quais situações é interessante utilizarmos esta nova estrutura.

6.1 Introdução

Em diversas situações, os tipos básicos de dados (inteiro, real, caractere,...) não são suficientes para representar a informação que se deseja armazenar. Por exemplo, se desejarmos armazenar uma quantidade grande

de números, fica inviável fazê-lo com uma variável para cada valor. Nesta situação, é interessante usar outras estratégias para armazenar este tipo de valor.

Existe a possibilidade de construção de novos tipos de dados a partir da composição (ou abstração) de tipos de dados primitivos. Neste capítulo introduziremos o conceito de *vetores*, mostrando em que situação este tipo de estrutura deve ser utilizado e quais as vantagens encontradas em sua utilização.

6.2 Motivação

Para entendermos a necessidade da utilização de vetores, observe o problema a seguir.

Problema - Como poderia ser feito um algoritmo para ler as 50 notas de uma turma e calcular sua média?

Solução - Este problema pode ser solucionado utilizando uma estrutura de repetição como o exemplo de código a seguir.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int i; // variável de controle
6     float nota, media, soma = 0;
7     for(i = 0; i < 50; i++)
8     {
9         printf("Digite uma nota:");
10        scanf("%f", &nota);
11        soma = soma + nota;
12    }
13    media = (float) soma / 50.0;
14    printf("Media = %f\n", media);
15 }
```

Considere agora fazer um algoritmo para ler as 50 notas de uma turma e calcular a sua média. Este algoritmo deverá imprimir também as notas lidas juntamente com a média da turma como na tabela a seguir.

Nota	Media
8.0	7.75
4.6	7.75
2.3	7.75
7.8	7.75
9.0	7.75
...	...

Veja que com o que aprendemos até agora, teríamos que criar uma variável para cada nota, impossibilitando o uso de estruturas de repetição. Imagine se ao invés de 50 notas fossem 50.000? Ficaria impraticável construir um programa para tratar esta quantidade de dados. A seguir veremos formas de resolver este tipo de situação.

6.3 Variáveis Compostas Homogêneas

Quando uma determinada estrutura de dados for composta de variáveis com o mesmo tipo primitivo, tem-se um conjunto homogêneo de dados. Estas variáveis são chamadas de variáveis compostas homogêneas.

As variáveis compostas homogêneas unidimensionais são utilizadas para representar arranjos unidimensionais de elementos de um mesmo tipo. Em outras palavras, são utilizadas para representar **vetores** (ou *arrays* em inglês).

6.4 Vetores

A sintaxe para declaração de um vetor é apresentada a seguir.

```
1 <tipo> <identificador>[<quantidade de elementos>];
```

Cada um dos elementos de um vetor é referenciado individualmente por meio de um número inteiro. Este número é chamado de índice do vetor. Na tabela a seguir, a primeira linha representa os índices e a segunda linha os dados contidos em cada posição do vetor.

	0	1	2	3	4
dados	3.5	2.1	4.7	7.1	1.5

O acesso aos elementos do vetor é feito através de seu índice. Os índices são referenciados através de colchetes do lado direito do identificador do vetor. Os índices de um vetor de tamanho n estão compreendidos entre 0 e $n-1$.

Veja no exemplo a seguir como poderíamos criar o vetor *dados* e atribuir os elementos da tabela vista anteriormente.

```
1 float dados[5];
2 dados[0] = 3.5;
3 dados[1] = 2.1;
4 dados[2] = 4.7;
5 dados[3] = 7.1;
6 dados[4] = 1.5;
```

Ao implementar vetores na linguagem C, alguns detalhes devem ser levados em consideração. Observe no código a seguir a criação dos vetores *v1*, *v2*, *v3* e *v4*.

```
1 int v1[5];
2 int v2[5] = {7};
3 int v3[5] = {7, 8, -1};
4 int v4[] = {5, 6, 7}
```

Neste código, *v1* não está sendo inicializado. Consequentemente não sabemos quais valores estão contidos neste vetor. O vetor *v2* foi inicializado com um único valor. Nesta situação, o valor será atribuído à primeira posição do vetor e o restante será preenchido com zero. No vetor *v3*, as três primeiras posições foram devidamente preenchidas. Novamente, o restante do vetor será inicializado com zero. O tamanho do vetor pode ser omitido desde que o mesmo seja inicializado durante sua criação. Por exemplo, observe que o vetor *v4* foi inicializado com 3 números. Por não ter sido especificado um tamanho, automaticamente o vetor será criado com um tamanho compatível com o número de elementos de sua inicialização. É importante lembrar que um vetor não pode ser criado sem tamanho e sem inicialização.

O tamanho de um vetor pode ser representado por um número inteiro ou por uma constante. A utilização de constantes é vantajosa caso seja necessário alterar o tamanho do vetor no código, esta alteração é feita somente na definição da constante (veja o uso de constantes no exemplo 2 a seguir).

Exemplo 1

O algoritmo a seguir usa a estrutura de repetição *for* para inicializar com zeros os elementos de um vetor de inteiros de tamanho 10 e o imprime sob a forma de uma tabela.

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int n[10], i;
6     for(i = 0; i <= 9; i++)
7     {
8         n[i] = 0;
9     }
10    printf("Elemento  Valor\n");
11    for(i = 0; i <= 9; i++)
12    {
13        printf("%5d %8d\n", i, n[i]);
14    }
15    return 0;
16 }
```

Elemento	Valor
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Exemplo 2

O algoritmo a seguir inicializa os dez elementos de um vetor *s* com os valores 2, 4, 6, ..., 20 e imprime o vetor em um formato de tabela.

```

1 #include <stdio.h>
2 #define TAMANHO 10
3
4 int main()
5 {
6     int s[TAMANHO], j;
7     for(j = 0; j <= TAMANHO - 1; j++)
8     {
9         s[j] = 2 + 2 * j;
10    }
11    printf("Elemento  Valor\n");
12    for(j = 0; j <= TAMANHO - 1; j++)
13    {
14        printf("%5d %8d\n", j, s[j]);
15    }
16    return 0;
17 }
```

Elemento	Valor
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20

6.5 Vetores e Funções

Na linguagem adotada neste livro, vetores são passados sempre por referência, ou seja, as modificações feitas na função refletem nos dados do vetor passado como parâmetro.

No exemplo a seguir é apresentado a função *imprimeVetor* que imprime um vetor de tamanho *tam*.

```
1 #include <stdio.h>
2 #define TAMANHO 10
3
4 void imprimeVetor(int vet[], int tam)
5 {
6     int i;
7     for(i = 0; i <= tam - 1; i++)
8     {
9         printf("%d\n", vet[i]);
10    }
11 }
12
13 int main()
14 {
15     int s[TAMANHO], i;
16     for(i = 0; i <= TAMANHO - 1; i++)
17     {
18         printf("Informe o valor do vetor na posicao %d: ", i);
19         scanf("%d", &s[i]);
20     }
21     imprimeVetor(s, TAMANHO);
22 }
```

6.6 Exercícios Resolvidos

Veremos a seguir alguns problemas envolvendo vetores com suas respectivas soluções.

Problema 1

Desenvolver uma função que receba um vetor de números reais e seu tamanho e retorne o índice do maior valor contido neste vetor. Se houver mais de uma ocorrência do maior valor, retornar o índice da primeira. Faça um programa para testar a função.

Solução

A solução para este problema consiste em, dentro da função proposta, inicializar uma variável auxiliar com o primeiro elemento do vetor e uma segunda variável com o índice 0. A seguir, dentro de uma estrutura de repetição, devemos comparar todos os demais elementos, um a um, com esta variável, atualizando-a sempre que encontrar um valor maior. Devemos atualizar também o índice sempre que isso acontecer.

```
1 #include <stdio.h>
2
3 int encontraMaior(float vet[], int tam)
4 {
5     int indice, i;
6     float maior = vet[0];
7     indice = 0;
8
9     for(i = 1; i < tam; i++)
10    {
11        if(vet[i] > maior)
12        {
13            maior = vet[i];
14            indice = i;
15        }
16    }
17    return indice;
18 }
19
20 int main()
21 {
22     float vetor[6] = {3.0, 4.3, 5.6, 2.8, 7.9, 3.4};
23     int posicao;
24     posicao = encontraMaior(vetor, 6);
25     printf("Maior valor esta na posicao %d.\n", posicao);
26 }
```

Problema 2

Criar uma função que receba um vetor de números reais e um valor inteiro representando o seu tamanho. Essa função deverá ordenar o vetor em ordem crescente.

Solução Proposta

Uma das soluções para este problema é usar uma estrutura de repetição dentro de outra. Na estrutura mais externa, o índice é incrementado da

primeira à penúltima posição do vetor. A estrutura interna vai no sentido contrário, comparando todos os elementos do vetor, dois a dois, e trocando a posição dos números quando o elemento de índice n for menor que o elemento de índice $n-1$. O limite desta segunda estrutura também é diferente. Devemos, a cada interação da estrutura externa, limitar a busca da estrutura interna. Na solução proposta, esta redução é obtida através da comparação dentro da estrutura de repetição ($j > i$). Neste caso, quanto maior for o índice i , menor será a quantidade de elementos a serem verificados.

```
1 #include <stdio.h>
2
3 void ordena(float vet[], int tam)
4 {
5     int i, j;
6     float aux;
7     for(i = 0; i < tam-1; i++)
8     {
9         for(j = tam-1; j > i; j--)
10        {
11            if (vet[j] < vet[j-1])
12            {
13                aux = vet[j];
14                vet[j] = vet[j-1];
15                vet[j-1] = aux;
16            }
17        }
18    }
19 }
20
21 int main()
22 {
23     int i;
24     float vet[5] = {11.0, 22.0, 3.0, 44.0, 5.0};
25     ordena(vet, 5);
26     for(i = 0; i < 5; i++)
27     {
28         printf("%.2f\n", vet[i]);
29     }
30     return 0;
31 }
```

6.7 Exercícios

Para cada problema a seguir, faça um algoritmo em C que o solucione. Se for pedido para desenvolver uma função para resolver o problema, crie também uma função principal que faça uso desta função.

1. Fazer um algoritmo para ler um vetor de números reais de tamanho 6 e imprimir a média aritmética dos elementos deste vetor.
2. Desenvolver um algoritmo para ler um vetor de números reais e um escalar. Após a leitura completa, imprimir o resultado da multiplicação deste vetor pelo escalar.
3. Dada uma tabela contendo a idade de 10 alunos, fazer um algoritmo para calcular o número de alunos com idade superior a média.
4. Fazer um algoritmo para ler e somar dois vetores de 10 elementos inteiros. Imprimir ao final os valores dessa soma.
5. Fazer um algoritmo para ler 20 valores do tipo inteiro e determinar qual o menor valor existente neste vetor e imprimir seu valor e índice.
6. Refazer o exercício anterior passando o vetor e seu tamanho como parâmetro para uma função e imprimir o menor valor do vetor e seu índice.
7. Fazer uma função para receber um vetor de números inteiros, seu tamanho e um valor a ser procurado neste vetor. A função deve retornar o número de ocorrências deste valor no vetor.
8. Fazer um algoritmo para ler um conjunto de 20 valores e armazená-los em um vetor V . A seguir, particionar V em dois outros vetores, P e I , conforme os valores de V forem pares ou ímpares, respectivamente. Ao final, imprimir os valores dos 3 vetores.
9. Fazer um algoritmo para ler um vetor de valores inteiros e imprimir na ordem crescente. O vetor deve ter tamanho N (utilizar a diretiva `#define`).
10. Dada uma tabela com as notas de uma turma de 20 alunos, fazer funções que retornem:
 - (a) A média da turma.
 - (b) A quantidade de alunos aprovados (≥ 60).
 - (c) A quantidade de alunos reprovados (< 60).

7 Vetores de Caracteres



Maria ficou responsável por preparar o aniversário surpresa de sua irmã mais nova. Para tal, Maria comprou bolo, refrigerantes e fez umas bandeirinhas com a palavra PARABÉNS para sua irmã. Ao chegar em casa, sua irmã ficou tão feliz com a surpresa que nem viu as bandeirinhas feitas por Maria.

Supondo que quiséssemos armazenar em um algoritmo a palavra que Maria escolheu para o aniversário de sua irmã, como poderíamos fazer? Com o que vimos até agora, poderíamos criar uma variável para cada letra. Funcionaria. Mas se o texto a ser armazenado for maior, fica complicado criar uma variável para cada letra. Como visto no capítulo anterior, aprendemos que é possível criar uma única estrutura para armazenar vários números. Esta mesma estrutura pode ser utilizada para armazenar caracteres - são vetores de caracteres. Neste capítulo aprenderemos como utilizar estes vetores em seus algoritmos.

7.1 Cadeia de Caracteres

Uma cadeia de caracteres é uma sequência de caracteres justapostos e são fundamentais no desenvolvimento de programas computacionais. São exemplos de cadeias de caracteres:

- Mensagem de e-mail;
- Texto de um programa;
- Nome e endereço em cadastro de clientes, alunos, etc...;
- Sequência genética. Um gene (ou o DNA de algum organismo) é composto de sequências dos caracteres A, T, G e C (nucleotídeos).

Uma variável usada para armazenar um caractere é representada da forma a seguir.

```
1 char letra; // criação da variável 'letra' do tipo caractere
2 letra = 'a'; // atribuição da letra 'a' a esta variável
```

Se em uma variável do tipo caractere pode-se armazenar somente um caractere, então para armazenar “jose”, “carro” etc é necessário utilizar vetores do tipo caractere.

A sintaxe para declaração de vetores de caracteres é semelhante à sintaxe de vetores numéricos como mostrado a seguir.

```
1 char <identificador>[<quantidade de caracteres>];
```

Na declaração a seguir a variável “cidade” é um vetor que pode armazenar até 12 caracteres.

```
1 char cidade[12];
```

Todos os caracteres em C são codificados através de uma tabela denominada ASCII¹. Esta codificação define 128 caracteres, sendo que 95 podem ser impressos e são mostrados a seguir. Nesta tabela, o código 32 representa o caractere de espaço. Esta tabela não mapeia caracteres com acento por ser baseada na língua inglesa.

¹<http://pt.wikipedia.org/wiki/ASCII>

	0	1	2	3	4	5	6	7	8	9
30			s	!	"	#	\$	%	&	'
40	()	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[\]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~			

Podemos utilizar caracteres explícitos ou a sua posição na tabela ASCII da mesma forma. Podemos inclusive imprimir um determinado caractere como inteiro (imprimirá sua posição na tabela) ou como caractere. Para ilustrar esta ideia, veja o exemplo a seguir.

```
1 char ch1 = 97;
2 char ch2 = 'B';
3 printf("int: %d char: %c \n", ch1, ch1);
4 printf("int: %d char: %c \n", ch2, ch2);
```

O conteúdo das variáveis *ch1* e *ch2* será impresso como inteiro e caractere como representado a seguir.

```
1 int: 97  char: a
2 int: 66  char: B
```

Em muitos problemas é mais fácil utilizar a constante do caractere entre apóstrofo ao invés de seu código.

7.2 *Strings*

Em C existe um tipo especial de cadeia de caracteres denominada *string*. Este tipo de cadeia termina, obrigatoriamente, com o caractere nulo: '\0' (\zero). Portanto, deve-se reservar uma posição para este caractere ao criar um vetor de caracteres do tipo *string*.

A inicialização de vetores de caracteres e *strings* são feitas de forma diferente. Veja a seguir exemplos de declaração e inicialização de vetores de caracteres normais.

```
1 char graus[5] = {'A','B','C','A','D'};  
2 char bin[8] = {0,'1',0,'1','1','1',0,'1'};
```

Podemos inicializar *strings* inserindo o caractere ‘\0’ ao seu final ou criando a *string* entre aspas. Veja a seguir alguns exemplos de declaração e inicialização de *strings*.

```
1 char disc[40] = {'A','l','g','o','r','i','t','m','o','\0'};  
2 char pais[10] = "Brasil";  
3 char nome[40] = "Joao da Silva";  
4 char cidade[] = "Juiz de Fora";
```

Observe no exemplo anterior que o tamanho da variável *cidade* não foi especificado. Neste caso, a variável terá o tamanho do texto mais um (para acomodar o ‘\0’). Só podemos criar vetores de caracteres sem tamanho se estes vetores forem inicializados durante a sua declaração.

7.3 Comandos de Leitura e Impressão

A leitura e a impressão de vetores de caracteres pode ou não ser feita caractere a caractere. Uma possibilidade de realizar a leitura e a escrita caractere a caractere é através dos comandos *getchar* e *putchar*, respectivamente. Veja a seguir um exemplo da utilização destes comandos para a leitura de um único caractere. Neste exemplo há a leitura de um vetor de caracteres chamado *placa* e sua posterior escrita.

```
1 #include <stdio.h>  
2  
3 int main()  
4 {  
5     char placa[7];  
6     int i;  
7     printf("Informe os caracteres da placa do seu carro: ");  
8     for(i = 0; i < 7; i++)  
9     {  
10         placa[i] = getchar();  
11     }  
12     printf("Placa informada: ");  
13     for(i = 0; i < 7; i++)  
14     {  
15         putchar(placa[i]);  
16     }  
17     return 0;  
18 }
```

Podemos utilizar a função *scanf* para fazer a leitura de vetores de caracteres. Contudo, o *scanf* é limitado pois a leitura é interrompida ao encontrar um caractere de espaço (' '), tabulação ('\t') ou nova linha ('\n'). Assim, se for digitado "Rio de Janeiro", a variável *s* conterá apenas "Rio". Veja a seguir um exemplo de código utilizando a função *scanf*.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char s[20];
6     printf("Digite uma string: ");
7     scanf("%s", s); // sem & antes de s
8     printf("String digitada: %s", s);
9     return 0;
10 }
```

Perceba no código anterior que não é necessário o *&* antes da variável *s* ao lermos vetores com o comando *scanf*.

Para leitura e escrita de *strings*, as funções mais adequadas são *gets* e *puts*, respectivamente. Veja a seguir um exemplo de utilização destas funções. Basicamente o código faz a leitura e escrita de uma *string* de até 20 caracteres.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char s[20];
6
7     printf("Digite uma string: ");
8     gets(s);
9
10    printf("String digitada: ");
11    puts(s);
12
13    return 0;
14 }
```

Exemplo 1

O algoritmo a seguir imprime uma cadeia de caracteres, caractere a caractere.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char s[30];
6     int i;
7     printf("Digite uma string: ");
8     gets(s);
9
10    // o código a seguir é equivalente a printf("%s",s);
11    for(i = 0; s[i] != '\0'; i++)
12        printf("%c",s[i]);
13
14    return 0;
15 }
```

Exemplo 2

O algoritmo a seguir calcula e imprime o comprimento (número de caracteres) de uma cadeia.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char s[30];
6     int i, n = 0;
7     printf("Digite uma string: ");
8     gets(s);
9
10    for(i = 0; s[i] != '\0'; i++)
11    {
12        n++;
13    }
14
15    printf("\nTamanho de %s: %d",s,n);
16
17    return 0;
18 }
```

Exemplo 3

O algoritmo a seguir faz uma cópia de uma cadeia, fornecida pelo usuário, para outra.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char dest[50]; // string destino
6     char orig[50]; // string origem
7     int i;
8     printf("Digite uma string: ");
9     gets(orig);
10
11    // copia cada caractere de orig para dest
12    for(i = 0; orig[i] != '\0'; i++)
13    {
14        dest[i] = orig[i];
15    }
16
17    // coloca o caractere nulo para marcar o fim da string
18    dest[i] = '\0';
19    puts(dest);
20    return 0;
21 }
```

Vale mencionar que existem várias funções em C para manipulação de *strings*. Essas funções estão declaradas no arquivo *string.h*. Entre elas, podemos destacar:

- `strcpy(char destino[], char origem[])` - Copia a *string origem* na *string destino*.
- `strlen(char str[])` - Retorna o tamanho da *string str*.
- `strcat(char destino[], char origem[])` - Faz concatenação (junção) da *string origem* com a *destino*. O resultado é armazenado na *string destino*.

7.4 Exercícios Resolvidos

Veremos a seguir alguns problemas envolvendo vetores de caracteres com suas respectivas soluções.

Problema 1

Criar uma função que receba como parâmetro uma cadeia de caracteres, seu tamanho e um segundo caractere. A função retornará a quantidade de vezes que o caractere procurado foi encontrado na cadeia.

Solução

É necessário percorrer a cadeia de caracteres usando uma estrutura de repetição e contar quantos são iguais ao caractere procurado, caractere a caractere.

```
1 #include <stdio.h>
2
3 int conta(char cadeia[], int tam, char procurado)
4 {
5     int encontrados, i;
6     i = 0;
7     encontrados = 0;
8
9     while(i < tam)
10    {
11        if(cadeia[i] == procurado)
12        {
13            encontrados = encontrados + 1;
14        }
15        i = i + 1;
16    }
17
18    return encontrados;
19 }
```

Problema 2

Criar uma função para verificar se a *string* *s2* está contida na *string* *s1*. A função retornará 1 se encontrar a *string* ou 0, caso contrário.

Ex.: Se *s1* fosse “Ana Maria Silva” e *s2* fosse “Maria”, a função retornaria 1, pois *s2* está contido em *s1*.

Solução

Uma das possíveis soluções envolve a utilização de duas estruturas de repetição aninhadas. A estrutura mais externa será utilizada para navegar sobre a *string* *s1* enquanto a estrutura interna comparará cada caractere de *s2* a partir da posição atual de *s1*.

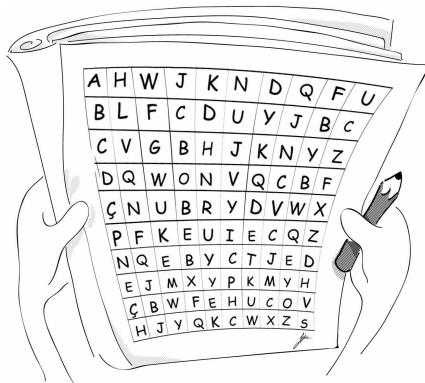
```
1 #include <stdio.h>
2 #include <string.h>
3
4 int buscaString(char s1[], char s2[])
5 {
6     int i, j, aux, tam1, tam2;
7     tam1 = strlen(s1);
8     tam2 = strlen(s2);
9     for(i = 0; i < tam1; i++)
10    {
11        aux=i;
12        for(j = 0; j < tam2 && aux < tam1; j++)
13        {
14            if(s2[j] != s1[aux])
15            {
16                break;
17            }
18            aux++;
19        }
20        if(j == tam2)
21        {
22            return 1;
23        }
24    }
25    return 0;
26 }
```

7.5 Exercícios

Para cada problema a seguir, faça um algoritmo em C que o solucione. Se for pedido para desenvolver uma função para resolver o problema, crie também uma função principal que faça uso desta função.

1. Fazer um algoritmo para contar o número de espaços em branco de uma *string*.
2. Fazer uma função que receba como parâmetro uma *string*. Esta função deve retornar o número de vogais que a *string* possui.
3. Escrever um algoritmo para ler uma *string* (com mais de uma palavra) e fazer com que a primeira letra de cada palavra fique em maiúscula.
Exemplo:
Entrada: lab. de linguagem de programacao
Saída: Lab. De Linguagem De Programacao
4. Escrever uma função para receber uma *string* e contar quantos caracteres desta *string* são iguais a ‘a’ e substituí-los por ‘b’. A função retornará o número de caracteres modificados.
5. Criar uma função para receber uma *string* e invertê-la.
6. Fazer um algoritmo para determinar e imprimir uma *string* que será a concatenação de duas outras *strings* lidas.
7. Fazer uma função para receber uma *string* e imprimir uma estatística de seus caracteres. Isto é, a função imprimirá o percentual de vogais, consoantes e outros caracteres contidos na *string*.
8. Fazer um algoritmo para ler uma *string* e transferir as consoantes para um vetor e as vogais para outro. Ao final, imprimir cada um dos vetores criados.
9. Fazer uma função para receber uma *string* e um caractere qualquer. A função deve remover todas as ocorrências do caractere passado por parâmetro e retornar o número de remoções realizadas.
10. Escrever uma função para receber uma *string* e retornar 1 se a mesma for um palíndromo e zero caso contrário. Uma palavra é dita ser palíndromo se a sequência de seus caracteres da esquerda para a direita é igual a sequência de seus caracteres da direita para a esquerda.
Ex.: arara, asa.

8 Vetores Multidimensionais



Maria gosta muito de jogos e decidiu comprar um caderno de caça palavras. O objetivo deste jogo é encontrar palavras dentro de uma tabela repleta de caracteres, de acordo com certa temática. Se quiséssemos criar um jogo deste tipo em algoritmos teríamos que armazenar as letras no formato de tabela. Uma das possibilidades de armazenamento destas letras seria através de vetores multidimensionais, também conhecidos como matrizes.

8.1 Definição

Assim como os vetores unidimensionais, os vetores multidimensionais (ou matrizes) são estruturas de dados homogêneas. A principal diferença em relação aos vetores (unidimensionais) é que matrizes possuem uma ou mais dimensões adicionais.

Matrizes são utilizadas quando os dados homogêneos necessitam de uma estruturação com mais de uma dimensão. Como exemplo podemos citar a utilização de matrizes na criação de um jogo de xadrez (o tabuleiro é naturalmente bidimensional), criar uma estrutura para guardar caracteres de um livro (três dimensões: duas para representar os caracteres de uma página e uma terceira para indicar as páginas) ou na resolução de problemas matemáticos matriciais de uma forma geral.

8.2 Declaração e Atribuição

A sintaxe para declaração de matrizes é semelhante a de vetores unidimensionais. Consideraremos, porém, a quantidade de elementos das outras dimensões como visto a seguir.

```
1 <tipo> <identificador>[<dim1>][<dim2>], ..., [<dim n>];
```

Para matrizes bidimensionais teríamos a declaração a seguir.

```
1 <tipo> <identificador>[<qde_linha>][<qde_col>];
```

Veja a seguir um exemplo de criação de uma matriz 3 x 4 (linha 1). Neste exemplo os índices variam de 0 a 2 para as linhas e de 0 a 3 para as colunas, totalizando 12 elementos. Na linha 2 do mesmo exemplo, temos a criação de uma matriz tridimensional de números reais. Neste exemplo, os índices variam de 0 a 2 para a 1^a dimensão, 0 a 5 para a 2^a e 0 a 4 para a 3^a dimensão.

```
1 int mat[3][4];
2 float mat[3][6][5];
```

Considere uma matriz 3 x 3 denominada *mat1* mostrada a seguir.

	0	1	2
0	3	8	5
1	9	2	1
2	7	3	6

Exemplos de acesso aos elementos da matriz *mat1* (linhas 2 e 3) e de atribuição de valores a uma determinada posição desta matriz (linhas 4 e 5) são mostrados a seguir.

```
1 int a, b;
2
3 a = mat1[1][2];
4 b = mat1[0][0];
5
6 mat1[0][1] = 15;
7 mat1[0][2] = -3;
```

No caso de atribuição de valores a uma matriz de número reais chamada *num* teríamos o resultado a seguir.

```
1 float num[2][3];
2
3 num[0][0] = 3.6;
4 num[0][1] = 2.7;
5 num[0][2] = 1.5;
6 num[1][0] = 5.0;
7 num[1][1] = 4.1;
8 num[1][2] = 2.3;
```

Podemos fornecer valores de cada elemento de uma matriz durante sua declaração da mesma forma como é feito em vetores como demonstrado a seguir.

```
1 float num[2][3] = {{3.6, 2.7, 1.5}, {5.0, 4.1, 2.3}};
```

Como em vetores numéricos, podemos inicializar matrizes com um número menor de elementos do que sua capacidade. Neste caso, o restante dos elementos não inicializados receberá o valor 0. No exemplo a seguir, a matriz *val* recebe os valores 3 e 7 em sua primeira linha e as demais recebem 0. A matriz *n1* ilustrada na segunda linha demonstra como podemos inicializar uma matriz com todos os elementos zerados em sua declaração. Havendo alguma inicialização, o número de elementos a ser atribuído para a matriz não pode exceder sua capacidade. A atribuição de valores na matriz *n2* ilustrada na terceira linha causaria um erro de sintaxe por ter uma quantidade de valores atribuídos superior ao tamanho da matriz.

```
1 int val[5][2] = {{3, 7}};
2 int n1[4][4] = {{0}};
3 int n2[2][4] = {{32, 64, 27, 18, 95}, {12, 15, 43, 17, 67}};
```

Exemplo 1

O algoritmo a seguir inicializa com zero os elementos de uma matriz inteira *N* de 5 linhas e 4 colunas e efetua a impressão ao final.

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int n[5][4], i, j;
6     for(i = 0; i < 5; i++)
7     {
8         for(j = 0; j < 4; j++)
9         {
10             n[i][j] = 0;
11         }
12     }
13     printf("Matriz");
14     for (i = 0; i < 5; i++)
15     {
16         printf("\n\nLinha %2d\n", i);
17         for(j = 0; j < 4; j++)
18         {
19             printf("%d ", n[i][j]);
20         }
21     }
22     return 0;
23 }
```

Matriz

Linha 0
0 0 0 0Linha 1
0 0 0 0Linha 2
0 0 0 0Linha 3
0 0 0 0Linha 4
0 0 0 0

Exemplo 2

O algoritmo a seguir inicializa os elementos de uma matriz m com os valores iguais à soma dos índices de cada elemento e imprime cada valor.

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int m[3][2], i, j;
6     for(j = 0; j < 3; j++)
7     {
8         for(i = 0; i < 2; i++)
9         {
10             m[j][i] = i + j;
11             printf("j=%d i=%d val=%d\n\n",
12                   j, i, m[j][i]);
13         }
14     }
15     return 0;
16 }
```

j=0 i=0 val=0
j=0 i=1 val=1
j=1 i=0 val=1
j=1 i=1 val=2
j=2 i=0 val=2
j=2 i=1 val=3

8.3 Matrizes e Funções

Matrizes serão passadas para funções da mesma forma que vetores. Entretanto, apenas a primeira dimensão pode ser omitida, independente da quantidade de dimensões que tiver a matriz, como mostra o exemplo a seguir.

```
1 void imprimeMatriz(float m[] [3], int lin, int col)
```

ou

```
1 void imprimeMatriz(float m[3] [3], int lin, int col)
```

8.4 Exercícios Resolvidos

Veremos a seguir alguns problemas envolvendo matrizes com suas respectivas soluções.

Problema 1

Criar uma função que receba uma matriz 2×3 de números reais e retorne a média dos valores da matriz. Criar uma função principal que chame esta função e imprima a média.

Solução

Todos os valores da matriz serão acumulados em uma variável real. A função retornará o valor dessa variável dividido pelo número de elementos desta matriz.

```
1 #include <stdio.h>
2
3 float mediaMatriz(float m[2] [3])
4 {
5     int i, j;
6     float media = 0;
7     for(i = 0; i < 2; i++)
8     {
9         for(j = 0; j < 3; j++)
10        {
11            media += m[i] [j];
12        }
13    }
14    return media / 6.0;
15 }
```

```
16 int main()
17 {
18     float mat[2][3] = {{3.4, 5.6, 4.0}, {2.0, 1.1, 4.9}};
19     float media = mediaMatriz(mat);
20     printf("A media da matriz foi %.2f", media);
21     return 0;
22 }
```

Problema 2

Criar uma função que zere todos os elementos negativos de uma matriz 5 x 5 de números inteiros. Criar funções para ler e imprimir matrizes e uma função principal que chame as funções criadas. Criar uma constante N com o valor 5 utilizando a diretiva *define* para indicar a dimensão da matriz.

Solução

Neste exercício, ao percorrermos a matriz devemos testar, elemento a elemento, se o valor é menor que zero. Se o teste for positivo, devemos alterar este elemento.

```
1 #include <stdio.h>
2 #define N 5
3
4 void zeraMatriz(int mat[N][N])
5 {
6     int i, j;
7
8     for(i = 0; i < N; i++)
9     {
10         for(j = 0; j < N; j++)
11         {
12             if(mat[i][j] < 0)
13             {
14                 mat[i][j] = 0;
15             }
16         }
17     }
18 }
19
20 void leMatriz(int mat[N][N])
21 {
22     int i, j;
```

```
24     for(i = 0; i < N; i++)
25     {
26         for(j = 0; j < N; j++)
27         {
28             printf("[%d] [%d]: ", i, j);
29             scanf("%d", &mat[i][j]);
30         }
31     }
32 }
33
34 void imprimeMatriz(int mat[N][N])
35 {
36     int i, j;
37     printf("Matriz:\n");
38     for(i = 0; i < N; i++)
39     {
40         for(j = 0; j < N; j++)
41         {
42             printf("%4d", mat[i][j]);
43         }
44         printf("\n");
45     }
46 }
47
48 int main()
49 {
50     int m[N][N];
51     leMatriz(m);
52     zeraMatriz(m);
53     imprimeMatriz(m);
54     return 0;
55 }
```

8.5 Exercícios

Para cada problema a seguir, faça um algoritmo em C que o solucione. Se for pedido para desenvolver uma função para resolver o problema, crie também uma função principal que faça uso desta função.

1. Fazer um algoritmo para exibir a soma de duas matrizes quadradas 3 x 3. Criar uma função para ler uma matriz (será chamada duas vezes com parâmetros diferentes) e uma segunda função para imprimir a soma das matrizes passadas como parâmetro.
2. Fazer um algoritmo para ler uma matriz quadrada de tamanho 10 e uma função para inverter as linhas pelas colunas em uma segunda matriz de mesmo tamanho. Imprimir ao final a segunda matriz.
3. Fazer um algoritmo para receber uma matriz quadrada 5 x 5 e criar uma matriz identidade. Imprimir a matriz após sua inicialização em outra função.
4. Fazer um algoritmo para ler um vetor de dimensão 5 e uma matriz quadrada de dimensão 5. Criar uma função para multiplicar o vetor pela matriz. Imprimir o resultado.
5. Desenvolver um algoritmo para ler uma matriz de números reais, um escalar e uma função para calcular a multiplicação da matriz pelo escalar. Imprimir o resultado em uma segunda função.
6. Fazer um algoritmo para ler uma matriz quadrada de dimensão 10, uma função para encontrar o maior valor desta matriz e outra função para encontrar o menor valor. Imprimir os valores encontrados na função principal.
7. Fazer um algoritmo para ler uma matriz 6 x 3 e uma função para gerar duas matrizes 3 x 3, a primeira com as 3 primeiras linhas e a segunda com as restantes.
8. Fazer um algoritmo para ler uma matriz de caracteres 10 x 5. Criar uma função que receba esta matriz e um caractere *ch*. Esta função deve retornar o número de vezes que o caractere *ch* foi encontrado na matriz.
9. Criar uma matriz tridimensional onde as linhas indicam as notas de matemática, história e geografia em três provas de 10 alunos e criar uma função para verificar quantos alunos passaram em tudo, ou seja, os que tenham média aritmética ≥ 60 nas 3 disciplinas.

9 Estruturas



Maria conseguiu um emprego numa clínica médica e sua função é cadastrar pacientes na recepção desta clínica. Maria precisa preencher vários campos como nome, telefone e endereço de cada cliente. Cada novo paciente recebe um número e posteriormente Maria pode recuperar as informações dele através deste valor.

Com o que vimos até agora, seria custoso acessarmos as informações citadas anteriormente através de um número que indexaria várias matrizes de caracteres. Por exemplo, para armazenarmos 10 nomes, teríamos que criar uma matriz $10 \times n$, sendo n o número máximo de caracteres de cada nome. Seria mais interessante agruparmos todas as informações de um determinado paciente em um único local como uma ficha, por exemplo. Com isso, se criássemos um vetor dessas fichas, poderíamos fazer o acesso através de um índice único. Esse tipo de funcionalidade pode ser obtido através de um recurso denominado **estruturas**.

9.1 Estruturas de Dados Heterogêneas

Até agora foram vistas as estruturas de dados homogêneas como vetores, matrizes e *strings*. Nestas estruturas todos os elementos são de um único tipo. No entanto, em muitos casos, necessitamos armazenar um conjunto de informações relacionadas, formado por diversos tipos de dados. Alguns

exemplos são endereços, fichas com dados pessoais e dados de um produto.

Quando uma determinada estrutura de dados for composta por vários campos, sendo eles tipos primitivos ou não, temos um conjunto heterogêneo de dados. As variáveis criadas a partir destas novas estruturas são chamadas de variáveis compostas heterogêneas, estruturas ou *structs*.

9.2 Definição

Uma estrutura pode ser definida como uma coleção de uma ou mais variáveis relacionadas (campos), onde cada variável pode ser de um tipo distinto. A sintaxe para definir uma estrutura com n campos é mostrada a seguir. Na linha 9 deste mesmo exemplo temos a criação de uma variável a partir desta estrutura.

```
1 struct <nomeEstrutura>
2 {
3     tipo1 identificador1;
4     tipo2 identificador2;
5     ...
6     tipoN identificadorN;
7 };
8 ...
9 struct <nomeEstrutura> var1;
```

Veja no código a seguir um exemplo de criação de uma estrutura *pessoa* e de uma variável deste tipo chamada *pessoal*.

```
1 struct pessoa
2 {
3     char nome[200];
4     char cpf[12];
5     int idade;
6 };
7 ...
8 struct pessoa pessoal;
```

Observe que neste código de exemplo houve a necessidade de explicitamente colocar o nome reservado *struct* antes do identificador da estrutura durante a criação da variável. Da mesma forma, havendo a necessidade de passarmos a estrutura como parâmetro para uma função, a palavra *struct* deve ser utilizada. Uma das alternativas para que a criação e utilização de estruturas seja realizada de forma mais transparente na linguagem C é

através da utilização de redefinição de tipo. Esta redefinição é realizada através do comando *typedef* (*type definition*).

Veja no exemplo a seguir como ficaria o código anterior com a utilização do comando *typedef*.

```
1 typedef struct spessoa
2 {
3     char nome[200];
4     char cpf[12];
5     int idade;
6 }pessoa;
7 ...
8
9 pessoa pessoa1;
```

Outra forma de utilização deste comando é omitindo o nome que fica ao lado da palavra reservada *struct* como no exemplo a seguir.

```
1 typedef struct
2 {
3     float x, y;
4 }ponto;
5 ...
6
7 ponto p1, p2;
```

Neste capítulo todas as estruturas serão criadas com comandos de redefinição de tipo para facilitar a sua utilização.

É importante observar que a definição de um tipo estrutura deve ficar, preferencialmente, fora do programa (principal) e de qualquer função. A declaração de variáveis de uma determinada estrutura pode ser feita em qualquer ponto do código, sendo *local*, se criada dentro de uma função ou *global*, caso contrário.

9.3 Manipulação

Os campos ou membros de uma estrutura podem ser acessados usando o operador de acesso *ponto* (.) entre a variável da estrutura e o nome do campo. Estes campos podem ser usados da mesma forma como as variáveis de tipos primitivos.

No exemplo criado na seção anterior, se quiséssemos atribuir o valor 35 ao campo *idade* da variável *pessoa1*, usariamós a sintaxe a seguir.

```
1 pessoa1.idade = 35;
```

Os comandos a serem utilizados para lermos ou escrevermos nos campos das estruturas serão escolhidos de acordo com o tipo destes campos. Então, para leremos um campo do tipo inteiro, utilizaremos o comando *scanf*. Da mesma forma, se desejarmos ler um campo de uma estrutura que representa uma *string*, utilizaremos preferencialmente o comando *gets*.

Podemos criar estruturas compostas por outras estruturas, como mostrado no exemplo a seguir. Neste mesmo código, observe como o campo *CEP* poderia ser acessado através de uma variável *ficha1* do tipo *fichapessoal*.

```
1 typedef struct
2 {
3     char rua[50];
4     int numero;
5     char bairro[20];
6     char cidade[30];
7     char siglaEstado[3];
8     int CEP;
9 }tipoEndereco;
10
11 typedef struct
12 {
13     char nome[50];
14     int telefone;
15     tipoEndereco endereco;
16 }fichaPessoal;
17
18 ...
19
20 fichaPessoal ficha1;
21 ficha1.endereco.CEP = 35380070;
```

Uma das vantagens ao utilizarmos estruturas é a possibilidade de copiar toda a informação de uma estrutura para outra do mesmo tipo com uma atribuição simples, como mostrado a seguir. Este método só funciona corretamente com campos primitivos alocados estaticamente na estrutura (há a necessidade de tomar outros cuidados ao fazer cópia de elementos alocados dinamicamente, mas este assunto foge do escopo deste livro).

```
1 typedef struct
2 {
3     int x;
4     int y;
5 } coordenadas;
6
7 ...
8
9 coordenadas primeira, segunda;
10 primeira.x = 20;
11 primeira.y = 30;
12 segunda = primeira;
```

Exemplo 1

Desenvolver uma estrutura chamada *funcionario* contendo dois campos: *matricula* de 15 caracteres e *nome* de 100 caracteres. Na função principal criar uma variável do tipo *funcionario* e fazer a leitura e escrita de seus campos.

```
1 #include <stdio.h>
2
3 typedef struct
4 {
5     char matricula[15];
6     char nome[100];
7 } funcionario;
8
9 int main()
10 {
11     funcionario f1;
12     gets(f1.matricula);
13     gets(f1.nome);
14     puts("Informacoes de f1:\n");
15     puts(f1.matricula);
16     puts(f1.nome);
17
18     return 0;
19 }
```

Exemplo 2

Considere que um aluno possui um nome (200 caracteres) e 4 notas (reais) como campos de uma estrutura. Desenvolver um algoritmo para ler e imprimir o nome e as notas de um aluno.

```
1 #include <stdio.h>
2
3 typedef struct
4 {
5     char nome[40];
6     float nota[4];
7 }aluno;
8
9 int main()
10 {
11     aluno alunol;
12     int i;
13
14     gets(alunol.nome);
15     for(i = 0; i <= 3; i++)
16     {
17         scanf("%f", &alunol.nota[i]);
18     }
19
20     puts(alunol.nome);
21     for(i = 0; i <= 3; i++)
22     {
23         printf("%f\n", alunol.nota[i]);
24     }
25
26     return 0;
27 }
```

Da mesma forma como acontece com os tipos primitivos, uma variável do tipo estrutura pode ser usada como parâmetro de uma função. Esta passagem poderá ser feita por valor ou referência. Quando uma variável (e não um vetor) é passada por referência, o acesso aos campos de uma estrutura é feito através de uma *seta* (->) e não de um *ponto* (.).

Veja o funcionamento da passagem de estruturas como parâmetros nos exemplos a seguir.

Exemplo 3

Fazer uma função para ler os dados de uma turma e uma função para calcular a média das notas desta turma. Desenvolver também o programa principal para chamar as funções de leitura e de cálculo da média. Os dados de uma disciplina são turma (um caractere), nome da disciplina (100 caracteres) e as notas dos 40 alunos da turma (real).

```
1 #include <stdio.h>
2
3 typedef struct
4 {
5     char turma, nome[100];
6     float media, notas[40];
7 }disciplina;
8
9 void leDadosDisc(disciplina *d1, int n)
10 {
11     int i;
12     printf("Disciplina: ");
13     gets(d1->nome);
14     printf("Informe a turma: ");
15     scanf("%c", &d1->turma);
16     printf("Informe %d notas\n", n);
17     for(i = 0; i < n; i++)
18     {
19         scanf("%f", &d1->notas[i]);
20     }
21 }
22
23 float calcMedia(disciplina d1, int n)
24 {
25     float soma = 0;
26     int i;
27     for(i = 0; i < n; i++)
28     {
29         soma = soma + d1.notas[i];
30     }
31     return soma / n;
32 }
33
34 int main()
35 {
36     disciplina discl;
37     leDadosDisc(&discl, 40);
38     discl.media = calcMedia(discl, 40);
39     printf("Disciplina %s", discl.nome);
40     printf("Turma %c:\n", discl.turma);
41     printf("Media: %.2f", discl.media);
42     return 0;
43 }
```

9.4 Vetores de Estruturas

Os vetores de estruturas podem ser criados da mesma forma que os vetores de tipos primitivos. Os algoritmos apresentados até o momento só fizeram menção a uma única instância da estrutura. É importante ressaltar que é necessária a definição da estrutura antes de declarar um vetor deste tipo.

Veja no exemplo a seguir a criação de uma estrutura e de um vetor deste mesmo tipo. Perceba que o vetor foi inicializado durante a criação. Neste caso, os parâmetros de inicialização devem estar na mesma ordem que os itens da estrutura.

```
1 typedef struct
2 {
3     int codigo;
4     char descricao[120];
5 }produto;
6
7 produto estoque[3] = {235,"Teclado", 245,"Mouse", 515,"Cabo"};
```

Podemos separar os campos da inicialização do vetor com chaves, como mostra o código a seguir.

```
1 produto estoque[3] = { {235, "Teclado"}, 
2                               {245, "Mouse"}, 
3                               {515, "Cabo"} };
```

Se forem inicializados menos elementos do que os alocados em memória, o restante dos campos do vetor ficará “zerado” (se numérico) ou vazio (se de caracteres).

9.5 Exercícios Resolvidos

Veremos a seguir alguns problemas envolvendo estruturas com suas respectivas soluções.

Problema 1

Considere que você está fazendo um programa que lê o nome e as 4 notas escolares de 8 alunos. Criar funções para ler e imprimir as informações destes alunos.

Solução

Será criado um vetor de estruturas na função principal sendo este vetor passado como parâmetro para as funções de leitura e impressão. Como há a necessidade de ler um vetor com as notas de cada aluno, será necessário utilizar uma estrutura de repetição, tanto na leitura quanto na impressão.

```
1 #include <stdio.h>
2
3 typedef struct
4 {
5     char nome[40];
6     float notas[4];
7 }aluno;
8
9 void leVetorAlunos(aluno a[], int n)
10 {
11     int i, j;
12     for(i = 0; i < n; i++)
13     {
14         printf("Informe o nome do aluno: ");
15         gets(a[i].nome);
16         printf("Informe as 4 notas: ");
17         for(j = 0; j < 4; j++)
18         {
19             scanf("%f%c", &a[i].notas[j]);
20         }
21     }
22 }
23
24 void imprimeVetorAlunos(aluno a[], int n)
25 {
26     int i, j;
27     printf("\nAlunos: ");
28     for(i = 0; i < n; i++)
29     {
30         printf("\n%20s ", a[i].nome);
31         for(j = 0; j < 4; j++)
32             printf("%.2f ", a[i].notas[j]);
33     }
34 }
35
36 int main()
37 {
38     aluno alunos[8];
39     leVetorAlunos(alunos, 8);
40     imprimeVetorAlunos(alunos, 8);
41     return 0;
42 }
```

Problema 2

Fazer um sistema para gerenciar um servidor de vídeos. Cada vídeo conterá título (200 caracteres), duração em segundos (inteiro) e número de visualizações (inteiro). A estrutura conterá também um identificador (*id*) inteiro para uso interno. Este identificador será único e sequencial, iniciando com o valor 1. O sistema armazenará um máximo de 1000 vídeos e permitirá a inclusão de vídeos (um a um), impressão dos vídeos cadastrados e das informações do vídeo mais visualizado. Para este sistema, crie um menu com estas 3 opções e implemente-as através de funções. Criar na função principal um vetor com 1000 vídeos e inicialize os *ids* dos vídeos com o valor 0. Utilizar este valor para definir se uma posição do vetor já foi utilizada.

Solução

Para facilitar o desenvolvimento deste exercício podemos usar a diretiva *define* criando uma constante de valor 1000. O menu pode ser feito com o comando *switch* e nele serão chamadas três funções, uma para incluir os vídeos, uma para imprimir todos os vídeos cadastrados e uma terceira para imprimir as informações do vídeo mais visualizado. Em todas as funções o identificador *id* será testado para saber qual posição do vetor foi utilizada.

```
1 #include <stdio.h>
2 #define MAX 1000
3
4 typedef struct
5 {
6     int id;
7     char titulo[200];
8     int duracao; // em segundos
9     int views;
10 }video;
11
12 void insereVideo(video videos[MAX])
13 {
14     int i;
15     for(i = 0; i < MAX; i++)
16     {
17         if(!videos[i].id) // encontra primeira posição vazia
18         {
19             videos[i].id = i+1;
20             printf("Dados sobre o video %d\n", videos[i].id);
21             printf("Informe o titulo: ");
22             gets(videos[i].titulo);
23             printf("Informe a duracao em segundos: ");
```

```
24         scanf("%d", &videos[i].duracao);
25         printf("Informe o numero de visualizacoes: ");
26         scanf("%d", &videos[i].views);
27         break; // insere apenas um video
28     }
29 }
30 }
31
32 void imprimeVideos(video videos[MAX])
33 {
34     int i;
35     printf("\nInformacoes sobre os videos cadastrados:\n");
36     for(i = 0; i < MAX; i++)
37     {
38         if(videos[i].id)
39         {
40             printf("%03d - %s\n", videos[i].id,videos[i].titulo);
41             printf("Duracao (seg): %d\n", videos[i].duracao);
42             printf("Visualizacoes: %d\n\n", videos[i].views);
43         }
44     }
45 }
46
47 void imprimeMaisVisualizado(video videos[MAX])
48 {
49     int i;
50     int maior = 0, selecionado = -1;
51     for(i = 0; i < MAX; i++)
52     {
53         if(videos[i].id)
54         {
55             if(videos[i].views > maior)
56             {
57                 maior = videos[i].views;
58                 selecionado = i;
59             }
60         }
61     }
62     if(selecionado != -1)
63     {
64         i = selecionado;
65         printf("Informacoes do video mais visualizado:\n");
66         printf("%d - %s\n", videos[i].id, videos[i].titulo);
67         printf("Duracao (seg): %d\n", videos[i].duracao);
68         printf("Visualizacoes: %d\n\n", videos[i].views);
69     }
70     else
71     {
72         printf("\nNenhum video cadastrado\n");
73     }
74 }
```

```
75
76 int main()
77 {
78     video videos[MAX] = { {0} }; // zera informações do vetor
79     int op;
80
81     do
82     {
83         printf("\nMenu\n");
84         printf("1 - Insere video\n");
85         printf("2 - Imprime videos cadastrados\n");
86         printf("3 - Imprime dados do video mais visualizado\n");
87         printf("0 - Sair\n");
88         printf("Opção: ");
89         scanf("%d%c", &op);
90         switch(op)
91         {
92             case 1:
93                 insereVideo(videos);
94                 break;
95             case 2:
96                 imprimeVideos(videos);
97                 break;
98             case 3:
99                 imprimeMaisVisualizado(videos);
100                break;
101            }
102        } while(op!=0);
103
104    return 0;
105 }
```

9.6 Exercícios

Para cada problema a seguir, faça um algoritmo em C que o solucione. Se for pedido para desenvolver uma função para resolver o problema, crie também uma função principal que faça uso desta função.

1. Fazer um algoritmo para ler os dados de um aluno. Os dados a serem guardados nesta estrutura são nome (100 caracteres), curso (40 caracteres) e idade (inteiro). Fazer a leitura e a impressão dos dados na função principal (*main*).
2. Criar uma estrutura chamada ponto contendo apenas as coordenadas *x* e *y* (real) do ponto. Na função principal declarar 2 pontos, ler as coordenadas *x* e *y* de cada ponto e calcular a distância entre eles. Apresentar no final a distância euclidiana entre estes dois pontos.
3. Fazer um algoritmo para ler as informações de *N* alunos (sendo *N* definido com a diretiva *define*). As informações que deverão ser lidas de cada aluno são matrícula (15 caracteres), nome (100 caracteres) e média final (real). Ao final, informar os nomes dos alunos que foram aprovados (média final ≥ 60). Para a leitura e a impressão usar funções.
4. Fazer um algoritmo para armazenar as informações de 11 jogadores de um time de futebol. Cada jogador possui nome (100 caracteres), número da camisa (inteiro), peso (real) e altura (real). Ler as informações de cada jogador e imprimir ao final estas informações, a inicial do jogador mais baixo e o número do jogador mais pesado. As operações solicitadas serão implementadas em funções.
5. Fazer um algoritmo para cadastrar os veículos de uma empresa. Poderão ser cadastrados no máximo 50 veículos contendo nome do condutor (100 caracteres), placa do veículo (8 caracteres), cor do veículo (20 caracteres) e turno de operação (1 caractere). O valor da variável turno poderá assumir os valores: *m* (manhã), *t* (tarde), *n* (noite) ou *i* (dia inteiro). Informar quais veículos foram cadastrados no sistema e quantos veículos estão em operação em cada turno. Utilizar funções para fazer as operações pedidas.
6. Fazer um algoritmo para gerenciar o estoque de uma empresa. Cada produto terá um identificador (inteiro), nome (150 caracteres), quantidade em estoque (inteiro) e preço unitário (real). Criar um vetor com 1000 produtos e inicializar o identificador de cada produto com

o valor 0. Criar um menu que permita inserir um novo produto (verificar o primeiro identificador com valor zero), remover um produto através de seu identificador, imprimir quantos produtos foram cadastrados, imprimir qual produto está com maior e menor estoque e imprimir as informações do produto com maior valor na empresa (multiplicar o valor unitário pelo estoque para descobrir o produto mais valioso).

7. Fazer um algoritmo para cadastrar pessoas. Cada pessoa conterá um CPF (12 caracteres), nome (100 caracteres), idade (inteiro), cidade (40 caracteres) e telefone (20 caracteres). Seu sistema armazenará no máximo 100 pessoas e permitirá ler as informações de uma pessoa (incluir a pessoa na primeira posição vaga do vetor), imprimir por idade (informar as idades mínima e máxima e imprimir as pessoas que estão neste intervalo), imprimir por inicial (pedir aqui somente a inicial do nome) e imprimir todos os registros cadastrados. Criar um menu para ter acesso às funções que realizarão as operações pedidas. Inicializar o CPF de todos os 100 registros no início do programa com o texto “vazio”. Utilizar essa informação para definir se um registro foi ou não lido naquela posição específica do vetor.

A Arquivos



Maria está trabalhando em uma empresa onde há várias informações que ainda não estão informatizadas. Desta forma, Maria cataloga todas estas informações em um arquivo com gavetas. Sempre que Maria precisa recuperar uma dessas informações, ela vai até o arquivo e procura nas gavetas a pasta correta, sendo este um trabalho bastante demorado.

Se quisermos fazer sistemas que permitam o acesso a informações a qualquer tempo, estas informações devem estar armazenadas. Uma das possíveis formas de armazenamento é através de arquivos. Quando ouvimos uma música em MP3 ou assistimos a um filme em DVD, o que estamos fazendo é acessando um arquivo previamente gravado com aquele tipo de informação. Neste apêndice serão apresentadas algumas formas de acessar e gravar dados em arquivos, com ênfase em gravação e leitura de arquivos no formato de texto.

Este apêndice trata de técnicas de persistência de dados. Persistir um dado significa armazená-lo em um dispositivo não volátil, isto é, um meio físico

recuperável como um arquivo ou um banco de dados. Até agora, toda a informação tratada era armazenada na memória RAM que é um meio volátil. Então, sempre que parávamos a execução dos programas, toda a informação que estava sendo trabalhada era perdida.

Agora, faremos acesso a dispositivos não voláteis como, por exemplo, um disco rígido, *pendrives* etc. Desta forma, a informação que está sendo trabalhada será armazenada e recuperada em futuras execuções de um determinado programa. Para facilitar a leitura deste capítulo, sempre que fizermos referência a um dispositivo de armazenamento não volátil, utilizaremos o termo disco.

A.1 Abertura e Fechamento de Arquivos

Toda manipulação de arquivos em C é feita através de um ponteiro para arquivo. Desta forma, criaremos um ponteiro para o tipo *FILE* definido no arquivo cabeçalho *stdio.h*. Podemos declarar um ponteiro de arquivo da forma a seguir.

```
1 FILE *p;
```

No fragmento de código acima, *p* é um ponteiro para um arquivo. Para trabalhar com arquivos precisamos associar o ponteiro visto anteriormente a um arquivo em disco. As funções *fopen* e *fclose* são específicas para abertura e fechamento de arquivos, respectivamente. A seguir, uma breve descrição destas funções.

Função *fopen()*

Esta é a função de abertura de arquivos com o protótipo a seguir.

```
1 FILE *fopen(char *nomeDoArquivo, char *modo);
```

A variável *nomeDoArquivo* determina qual arquivo em disco será aberto. Este nome deve ser válido no sistema operacional que estiver sendo utilizado. O modo de abertura diz à função *fopen* que tipo de uso será feito deste arquivo. A tabela a seguir mostra os valores de modo de abertura válidos.

Modo	Significado
<i>r</i>	Abre um arquivo de texto para leitura. O arquivo deve existir antes de ser aberto.
<i>w</i>	Abre um arquivo de texto para gravação. Se o arquivo não existir, ele será criado. Se já existir, o conteúdo anterior será apagado.
<i>a</i>	Abre um arquivo de texto para gravação. Os dados serão adicionados no fim do arquivo (<i>append</i>), se ele já existir, ou um novo arquivo será criado, caso contrário.

Para manipularmos arquivos binários, utilizaremos os modos *rb*, *wb* e *ab* da mesma forma como faríamos com arquivos de texto. É possível também criar arquivos de leitura e escrita através do modificador *+*. Desta forma, podemos usar os modos *r+*, *w+*, *a+*, *r+b*, *w+b* e *a+b*. Por exemplo, para abrir um arquivo binário para escrita, teríamos o seguinte fragmento de código.

```

1 FILE *fp;
2 // arquivo se chama exemplo.bin, localizado no diretório atual
3 fp = fopen("exemplo.bin", "wb");
4 if(!fp)
5 {
6     printf("Erro na abertura do arquivo.");
7 }
```

A estrutura condicional testa se o arquivo foi aberto com sucesso. Uma vez aberto um arquivo, podemos ler ou escrever dados utilizando as funções que serão apresentadas a seguir.

Função exit()

Aqui abrimos um parênteses para explicar a função *exit* que possui o protótipo a seguir.

```

1 void exit(int codigoDeRetorno);
```

Esta função está disponível no arquivo cabeçalho *stdlib.h*. Sua utilidade é abortar a execução do programa. Esta função pode ser chamada de qualquer ponto no código e faz com que o programa termine e retorne o *codigoDeRetorno* para o sistema operacional. A convenção mais usada é que um programa retorne zero no caso de um término normal e retorne um

número não nulo no caso de ter ocorrido um problema. A função *exit* se torna importante em códigos que fazem uso de alocação dinâmica e abertura de arquivos pois nestes casos, se o programa não conseguir a memória necessária ou não conseguir abrir o arquivo, a melhor saída pode ser terminar a execução deste programa.

Poderíamos reescrever o exemplo da seção anterior usando agora a função *exit* para interromper o programa em caso de falha.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main (void)
5 {
6     FILE *fp;
7     ...
8     fp = fopen("exemplo.bin", "wb");
9     if(!fp)
10    {
11         printf("Erro na abertura do arquivo.");
12         exit(1);
13     }
14     ...
15     return 0;
16 }
```

Função *fclose()*

Para fechar um arquivo devemos usar a função *fclose* como a seguir.

```
1 int fclose(FILE *fp);
```

O ponteiro *fp* passado à função *fclose* determina o arquivo a ser fechado. A função retorna zero no caso de sucesso.

Fechar um arquivo faz com que qualquer caractere que tenha permanecido no *buffer* associado ao fluxo de saída seja gravado em disco.

Mas o que é um *buffer*? Quando você envia caracteres para serem gravados em um arquivo, estes caracteres são armazenados temporariamente em uma área de memória (o *buffer*) ao invés de serem escritos em disco imediatamente. Quando o *buffer* estiver cheio, seu conteúdo é escrito no disco de uma vez. A razão para se fazer isto está relacionada com a eficiência nas leituras e gravações de arquivos. Se, para cada caractere que fôssemos

gravar, tivéssemos que posicionar a cabeça de gravação (supondo um disco mecânico) em um ponto específico do disco, apenas para gravar aquele caractere, as gravações seriam muito lentas. Assim, estas gravações somente serão efetuadas quando houver um volume razoável de informações a serem gravadas ou quando o arquivo for fechado.

A.2 Leitura e Escrita em Arquivos

Podemos utilizar algumas funções em C para ler e escrever em arquivos.

Função `getc`

A função *getc* retorna o código do caractere lido do arquivo *fp*.

```
1 int getc(FILE *fp);
```

Função `putc`

A função *putc* é a primeira função de escrita de arquivo que veremos. Esta função escreve um único caractere no arquivo. Ela recebe como parâmetros o código ASCII de um caractere e um ponteiro para o arquivo onde este caractere será gravado.

```
1 int putc(int ch, FILE *fp);
```

O programa a seguir lê uma *string* do teclado e a escreve, caractere por caractere, em um arquivo em disco (o arquivo chamado *arq.txt*, que será aberto no diretório corrente).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     FILE *fp;
7
8     char string[100];
9     int i;
10    fp = fopen("arq.txt", "w"); // arquivo ASCII, para escrita
11
12    if(!fp)
13    {
14        printf("Erro na abertura do arquivo");
```

```
15     exit(0);
16 }
17
18 printf("Entre com a string a ser gravada no arquivo: ");
19 gets(string);
20 for(i = 0; string[i]; i++)
21 {
22     putc(string[i], fp);
23 }
24 fclose(fp);
25
26 return 0;
27 }
```

Função fgets

Para se ler uma *string* a partir de um arquivo podemos usar a função *fgets*.

```
1 char *fgets(char *str, int tamanho, FILE *fp);
```

A função recebe como parâmetros uma *string* a ser lida, o limite máximo de caracteres a serem lidos e o ponteiro que está associado ao arquivo de onde a *string* será lida. A função lê a *string* até que um caractere de nova linha seja lido ou *tamanho-1* caracteres tenham sido lidos. Se o caractere de nova linha ('\n') for lido, ele fará parte da *string*, o que não acontecia com a função *gets*. A *string* resultante sempre terminará com '\0' (por isto somente *tamanho-1* caracteres, no máximo, serão lidos).

Função fputs

A função *fputs* escreve uma *string* *str* num arquivo *fp*.

```
1 char *fputs(char *str, FILE *fp);
```

Função fread

Com a função *fread* podemos escrever e ler blocos de dados. Para tanto, temos as funções *fread* e *fwrite*. O protótipo de *fread* é mostrado a seguir.

```
1 unsigned fread(void *buff, int bytes, int count, FILE *fp);
```

O *buffer* é a região de memória na qual serão armazenados os dados lidos. A variável *bytes* é o tamanho da unidade a ser lida e a variável *count*, a

quantidade desta unidade. Isto significa que o número total de *bytes* lidos é *bytes* * *count*.

A função retorna o número de unidades efetivamente lidas. Este número pode ser menor que *count* quando o fim do arquivo for encontrado ou ocorrer algum erro. Quando o arquivo for aberto para dados binários, *fread* pode ler qualquer tipo de dados.

Função fwrite()

A função *fwrite* funciona de modo similar à função *fread*, porém escrevendo no arquivo com o protótipo a seguir.

```
1 unsigned fwrite(void *buffer, int bytes, int count, FILE *fp);
```

A função retorna o número de itens escritos. Este valor será igual a *count* a menos que ocorra algum erro. O exemplo a seguir ilustra o uso de *fread* e *fwrite* para gravar e posteriormente ler uma variável *float* em um arquivo binário. Neste exemplo foi utilizado também o operador *sizeof*, que retorna o tamanho em *bytes* de uma variável ou de um tipo de dados.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     FILE *pf;
7
8     float pi = 3.1415;
9     float piLido;
10
11    // abre arquivo binário para escrita
12    if ((pf = fopen("arquivo.bin", "wb")) == NULL)
13    {
14        printf("Erro na abertura do arquivo");
15        exit(1);
16    }
17
18    if(fwrite(&pi, sizeof(float), 1, pf) != 1)
19    {
20        printf("Erro na escrita do arquivo");
21    }
22    // fecha o arquivo
23    fclose(pf);
24 }
```

```
25 // abre o arquivo novamente para leitura
26 if((pf = fopen("arquivo.bin", "rb")) == NULL)
27 {
28     printf("Erro na abertura do arquivo");
29     exit(1);
30 }
31
32 // lê em piLido o valor da variável armazenada
33 if (fread(&piLido, sizeof(float), 1, pf) != 1)
34 {
35     printf("Erro na leitura do arquivo");
36     exit(1);
37 }
38
39 printf("\nO valor de PI, lido do arquivo é: %f", pilido);
40 fclose(pf);
41 return(0);
42 }
```

A.3 Outros Comandos

Função feof()

A função *feof* verifica se um arquivo chegou ao seu fim. Esta função retorna não-zero se o arquivo chegou ao fim ou zero, caso contrário.

```
1 int feof(FILE *fp);
```

Outra forma de verificar se o final do arquivo foi atingido é comparar o caractere lido com a constante *EOF*. O programa a seguir abre um arquivo já existente e o lê, caractere a caractere, até que o final do arquivo seja atingido. Os caracteres lidos são apresentados na tela.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     FILE *fp;
7     char c;
8     fp = fopen("arquivo.txt", "r");
9     if(!fp)
10    {
11        printf( "Erro na abertura do arquivo");
12        exit(0);
13    }
```

```
14     while((c = getc(fp)) != EOF)
15     {
16         printf("%c", c); // imprime o caractere lido
17     }
18     fclose(fp);
19     return 0;
20 }
```

A seguir é apresentado um exemplo utilizando comandos de leitura, escrita, abertura e fechamento de arquivo.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 void main()
6 {
7     FILE *p;
8     char c, str[30], frase[80] = "Este e um arquivo chamado: ";
9     int i;
10
11    // lê um nome para o arquivo a ser aberto
12    printf("\n\n Entre com um nome para o arquivo: \n");
13    gets(str);
14
15    if (!(p = fopen(str, "w")))
16    {
17        printf("Erro! Impossivel abrir o arquivo!\n");
18        exit(1);
19    }
20
21    strcat(frase, str);
22    for (i = 0; frase[i]; i++)
23    {
24        putc(frase[i], p);
25    }
26    fclose(p);
27
28    p = fopen(str, "r"); // abre novamente para leitura
29    c = getc(p); // lê o primeiro caractere
30    while (!feof(p)) // enquanto não chegar ao final do arquivo
31    {
32        printf("%c", c); // imprime o caractere na tela
33        c = getc(p); // lê um novo caractere no arquivo
34    }
35    fclose(p); // fecha o arquivo
36    return 0;
37 }
```

Arquivos pré-definidos

Quando se começa a execução de um programa, o sistema automaticamente abre alguns arquivos pré-definidos:

stdin: dispositivo de entrada padrão (geralmente o teclado)

stdout: dispositivo de saída padrão (geralmente o vídeo)

stderr: dispositivo de saída de erro padrão (geralmente o vídeo)

stdaux: dispositivo de saída auxiliar (em muitos sistemas, associado à porta serial)

stdprn: dispositivo de impressão padrão (em muitos sistemas, associado à porta paralela)

Cada uma destas constantes pode ser utilizada como um ponteiro para arquivo, para acessar os periféricos associados a eles como nos exemplos a seguir.

```
1 ch = getc(stdin); // leitura de um caractere através do teclado
2 putc(ch, stdout); // impressão deste caractere na tela.
```

Funções *ferror()* e *perror()*

A função *ferror* retorna zero, se nenhum erro ocorreu e um número diferente de zero se algum erro ocorreu durante o acesso ao arquivo. Esta função é muito útil quando queremos verificar se cada acesso a um arquivo teve sucesso, de modo que consigamos garantir a integridade dos dados. Na maioria dos casos, se um arquivo pode ser aberto, ele pode ser lido ou gravado. Porém, existem situações em que isto não ocorre. Por exemplo, pode acabar o espaço em disco enquanto gravamos, ou o disco pode estar com problemas e não conseguimos ler, etc. Esta função tem o protótipo a seguir.

```
1 int ferror(FILE *fp);
```

Uma função que pode ser usada em conjunto com *ferror* é a *perror* (*print error*), cujo argumento é uma *string* que normalmente indica em que parte do programa o problema ocorreu. No exemplo a seguir, fazemos uso das funções *ferror* e *perror*.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main()
6 {
7     FILE *pf;
8     char string[100];
9     if ((pf = fopen("arquivo.txt", "w")) == NULL)
10    {
11        printf("\nNao consigo abrir o arquivo! ");
12        exit(1);
13    }
14    do
15    {
16        printf("\nDigite uma string.");
17        printf("\nPara terminar, digite <enter>: ");
18        gets(string);
19        fputs(string, pf);
20        putc('\n', pf);
21        if(ferror(pf))
22        {
23            perror("Erro na gravacao");
24            fclose(pf);
25            exit(1);
26        }
27    } while(strlen(string) > 0);
28    fclose(pf);
29 }
```

Sobre os Autores



Rodrigo Luis de Souza da Silva possui graduação em Ciência da Computação pela Universidade Católica de Petrópolis (1999), mestrado em Engenharia de Sistemas e Computação (2002), doutorado em Engenharia Civil (2006) pela Universidade Federal do Rio de Janeiro e Pós-doutorado em Ciência da Computação pelo Laboratório Nacional de Computação Científica (2008). Atualmente é professor no Departamento de Ciência da Computação da Universidade Federal de Juiz de Fora. Suas principais áreas de pesquisa são Realidade Aumentada, Realidade Virtual e Computação Gráfica.



Alessandreia Marta de Oliveira possui graduação em Matemática Bacharelado em Informática pela Universidade Federal de Juiz de Fora (1999) e mestrado em Engenharia de Sistemas e Computação pela Universidade Federal do Rio de Janeiro (2003). Atualmente é professora da Universidade Federal de Juiz de Fora e doutoranda em Computação pela Universidade Federal Fluminense. Suas principais áreas de pesquisa são Banco de Dados e Engenharia de Software.

Referências Bibliográficas

- UFMG. **Curso de Linguagem C**, 2005.
<http://www.ead.cpdee.ufmg.br/cursos/C/>, Acessado em 17 de maio de 2012.
- Cormen, T.; Leiserson, C.; Rivest, R. ; Stein, C. **Introduction to Algorithms**. 3a. ed., Editora Campus, 2012.
- Deitel, H.; Deitel, P. **C: Como Programar**. 6a. ed., Pearson Brasil, 2011.
- Guimarães, A. M.; Lages, N. A. C. **Algoritmos e Estruturas de Dados**. LTC, 1994.
- Kernighan, B.; Ritchie, D. **C: A Linguagem de Programação - Padrão ANSI**. Editora Campus, 1989.
- Schildt, H. **C Completo e Total**. 3a. ed., Prentice Hall, 2005.

Algoritmos em C

Este livro visa introduzir os conceitos básicos de lógica de programação com o objetivo de ensinar, através de exemplos, como construir algoritmos. A linguagem C foi escolhida para este material por ser a linguagem básica para diversas disciplinas em vários cursos de Ciência da Computação. É importante ressaltar que este livro não foca em detalhes da linguagem, mas sim em como utilizá-la para o entendimento dos conceitos básicos da construção de algoritmos.

No início do livro, o leitor encontrará capítulos sobre tipos de dados, variáveis e comandos de entrada e saída. Os códigos iniciais serão apresentados através da introdução do primeiro fluxo de controle: a sequência simples. Dando enfoque na programação modular, antes de abordarmos as demais estruturas de controle, o conceito de funções será apresentado. Após esta introdução, as estruturas condicionais e estruturas de repetição serão exploradas.

Ao dominar as estruturas básicas, o leitor será apresentado aos tipos de dados compostos que estão divididos em homogêneos (vetores e matrizes) e heterogêneos (registros). Para finalizar, um capítulo adicional com instruções sobre como manipular arquivos na linguagem C será apresentado.

Este livro é recomendado tanto para alunos interessados em aprender programação quanto para profissionais que buscam um material acessível para se iniciar no mundo da programação de computadores.

Apoio:



ISBN 978-85-917697-1-1

