

Linguagem de Programação C

Fundamentos da Linguagem C

A linguagem C é uma linguagem de alto nível, genérica. Foi desenvolvida por programadores para programadores, tendo como meta características de flexibilidade e portabilidade. O C é uma linguagem que nasceu juntamente com o advento da teoria de linguagem estruturada e do computador pessoal. Assim, tornou-se rapidamente uma linguagem “popular” entre os programadores. O C foi usado para desenvolver o sistema operacional UNIX, e hoje está sendo usada para desenvolver novas linguagens, entre elas a linguagem C++ e Java.

Histórico

1970: Denis Ritchie desenha uma linguagem a partir do BCPL nos laboratórios da Bell Telephones, Inc. Chama a linguagem de B.

1978: Brian Kerningham junta-se a Ritchie para aprimorar a linguagem. A nova versão chama-se C. Pelas suas características de portabilidade e estruturação já se torna popular entre os programadores.

~1980: A linguagem é padronizada pelo American National Standard Institute: surge o ANSI C.

~1990: A Borland International Company, fabricante de compiladores profissionais escolhe o C e o Pascal como linguagens de trabalho para o seu Integrated Development Enviroment (Ambiente Integrado de Desenvolvimento): surge o Turbo C.

~1992: O C se torna ponto de concordância entre teóricos do desenvolvimento da teoria de Object Oriented Programming (programação orientada a objetos): surge o C++.

Características

Entre as principais características do C, pode-se citar:

- O C pode ser considerado uma linguagem de médio nível, pois possui instruções que a tornam ora uma linguagem de alto nível e estruturada como o Pascal, ora uma linguagem de baixo nível como o Assembler, pois possui instruções muito próximas da máquina.
- O C é uma linguagem com uma sintaxe bastante estruturada e flexível, tornando sua programação bastante simplificada.
- Programas em C são compilados, gerando programas executáveis.
- O C compartilha recursos tanto de alto quanto de baixo nível, pois permite acesso e programação direta do microprocessador. Com isto, rotinas cuja dependência do tempo é crítica, podem ser facilmente implementadas usando instruções em Assembly. Por esta razão o C é a linguagem preferida dos programadores de aplicativos.
- O C é uma linguagem estruturalmente simples e de grande portabilidade. O compilador C gera códigos mais enxutos e velozes do que muitas outras linguagens.
- Embora estruturalmente simples (poucas funções intrínsecas), o C não perde funcionalidade, pois permite a inclusão de uma farta quantidade de rotinas do usuário. Os fabricantes de compiladores fornecem uma ampla variedade de rotinas pré-compiladas em bibliotecas.

Estrutura de um programa em C

Um programa em C é constituído de:

- um cabeçalho contendo as diretivas de compilador onde se definem o valor de constantes simbólicas, declaração de variáveis, inclusão de bibliotecas, declaração de rotinas, etc.
- um bloco de instruções principal e outros blocos de rotinas.
- documentação do programa: comentários.

Conjunto de caracteres

Um programa fonte em C é um **texto não formatado** escrito em um editor de textos ou em um ambiente de programação, como o Turbo C, da Borland, usando o conjunto padrão de caracteres ASCII. A seguir estão os caracteres utilizados em C. Caracteres válidos:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
1	2	3	4	5	6	7	8	9	0																	
+	-	*	/	\	=		&	!	?	#	%	()	{	}	[]	_	`	^	~	.	,	:	<	>

Alguns caracteres não válidos (válidos apenas em strings): @ \$ % ^ & * (e demais símbolos e caracteres acentuados)



Comentários

Em C, comentários podem ser escritos em qualquer lugar do programa para facilitar a sua posterior interpretação. Para que o comentário seja identificado como tal, ele deve ter um `/*` antes e um `*/` depois, no caso de querer comentar um bloco de instruções, ou `//` no caso de querer comentar linhas individuais. Exemplo:

```
/* esta é uma  
linha de  
comentário em C */
```

ou:

```
// esta linha está comentada
```

Dica: No Visual C++ da Microsoft, quando se quer fazer comentários, seja de uma ou várias linhas, basta utilizar a ferramenta , e para retirar os comentários basta utilizar .

Diretivas de Compilação

Em C, existem comandos que são processados durante a **compilação** do programa. Estes comandos são genericamente chamados de **diretivas de compilação**. Estes comandos informam ao compilador do C basicamente quais são as **constantes simbólicas** usadas no programa e quais **bibliotecas** devem ser anexadas ao programa executável. A diretiva `#include` diz ao compilador para incluir na compilação do programa outros arquivos. Geralmente estes arquivos contêm bibliotecas de funções ou rotinas do usuário. Exemplos de bibliotecas do C:

<code>stdio.h</code>	Funções de entrada e saída em disco
----------------------	-------------------------------------

<code>ctype.h</code>	Funções que tratam de caracteres
<code>String.h</code>	Funções de tratamento de strings
<code>mem.h</code>	Funções que manuseiam a memória
<code>math.h</code>	Funções matemáticas
<code>alloc.h</code>	Funções de alocação dinâmica de memória
<code>graphics.h</code>	Funções de manuseio do vídeo no modo gráfico
<code>conio.h</code>	Funções de manuseio do vídeo no modo texto
<code>time.h</code>	Funções de manuseio de data e hora

A diretiva `#define` diz ao compilador quais são as constantes simbólicas usadas no programa. Ambas as diretivas serão vistas a seguir.

Primeiro Exemplo

O exemplo a seguir apenas mostra no vídeo o texto `Alo Mundo!!!` e aguarda até uma tecla ser digitada. Todo programa em C consiste em uma ou mais funções. A única função que necessariamente precisa estar presente é a denominada `main()`, que é a primeira função a ser chamada quando a execução do programa começa.

```
/* Programa simples para mostrar Alo Mundo na tela */

#include <stdafx.h>

void main()
{
    printf("Alo Mundo!!!");
    getchar();
}
```

Observação: Como a ferramenta utilizada para os exemplos é o Visual C++ da Microsoft, algumas características são específicas desse compilador.

Percebe-se que a função `main()` está entre {}, indicando o início e o fim da função. O `printf` é responsável por mostrar o string `Alo Mundo!!!` na tela, sendo que todo string deve estar entre aspas, e o `getchar()` aguarda até que uma tecla seja pressionada. É importante frisar que o C diferencia caracteres minúsculos de maiúsculos, e que praticamente toda a programação é feita em minúsculo. O `#include <stdafx.h>` é necessário para compilar o programa. O `void` antes do `main` significa que a função não retorna valor, o que será discutido adiante.

Palavras Reservadas

Por ser capaz de manipular bits, bytes e endereços, C se adapta bem a programação a nível de sistema. E tudo isto é realizado por apenas 43 palavras reservadas no Turbo C, 32 nos compiladores padrão ANSI e 28 no C Padrão. O Visual C++ possui mais palavras reservadas. É importante saber que elas existem pois não podemos declarar uma variável, por exemplo, com o mesmo nome de uma palavra reservada. As palavras reservadas do C++ são:

```
asm, auto, bool, break, case, catch, char, class, const, const_cast,
continue, default, delete, do, double, dynamic_cast, else, enum, explicit,
extern, false, float, for, friend, goto, if, inline, int, long, mutable,
namespace, new, operator, private, protected, public, register,
reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct,
switch, template, this, throw, true, try, typedef, typeid, typename, union,
unsigned, using, virtual, void, volatile, wchar_t
```

Importante: A linguagem C é sensível ao contexto, o que significa dizer que um identificador escrito em letras minúsculas não é equivalente a outro com o mesmo nome mas escrito em letras maiúsculas. Assim, por exemplo, a variável RESULT não é a mesma que result nem que a variável Result. Então, é possível ter uma variável com nome Friend, por exemplo, sem infringir a regra de não utilização de mesmos identificadores que as palavras reservadas.

Tipos de Dados Padrão

A linguagem C trata vários tipos de dados padrão (ou fundamentais). A declaração de variáveis deve se basear em um destes tipos. Os tipos de dados são usados para definir tamanho em bytes que uma variável vai ocupar na memória e qual o intervalo de valores que poderá armazenar.

Tipos Inteiros

São tipos numéricos exatos, sem casas decimais.

Tipo	Tamanho em Bytes	Valor Mínimo	Valor Máximo
<code>unsigned short</code>	2	0	65535
<code>short</code>	2	-32.768	32.767
<code>unsigned int*</code>	2 ou 4	ver short e long	ver short e long
<code>int*</code>	2 ou 4	ver short e long	ver short e long
<code>unsigned long</code>	4	0	4.294.967.295
<code>long</code>	4	-2.147.483.648	2.147.483.647

* O tamanho de tipo de dado `int` depende no tamanho do tamanho da “palavra” do sistema operacional. Assim, no MS-DOS possui 16 bits, enquanto em sistemas de 32 bits (como Windows 9x/2000/NT) possui 32 bits (4 bytes). Em resumo, ele pode ser do mesmo tamanho que um `short`, ou que um `long`, dependendo da situação.

Tipos Reais

São tipos numéricos com casas decimais.

Tipo	Tamanho em Bytes	Valor Mínimo	Valor Máximo
<code>float</code>	4	3.4E-38	3.4E+38
<code>double</code>	8	1.7E-308	1.7E+308
<code>long double</code>	10	3.4E-4932	1.1E+4932

Tipos Texto

São tipos que tratam de caracteres.

Tipo	Tamanho em Bytes	Valor Mínimo	Valor Máximo
<code>unsigned char</code>	1	0	255
<code>char</code>	1	-128	127

Tipo Lógico

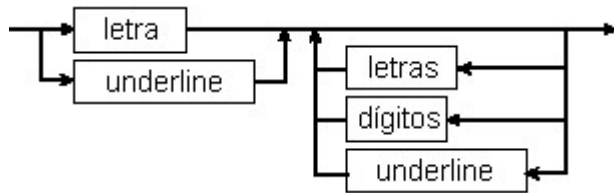
O tipo lógico existente é o `bool`, onde os valores possíveis são `true` ou `false`.

Tipo	Tamanho em Bytes	Valores Possíveis
------	------------------	-------------------

<code>bool</code>	<code>1</code>	<code>true</code> ou <code>false</code>
-------------------	----------------	---

Identificadores

São considerados identificadores os nomes dados para constantes, variáveis, funções, etc. No C, os identificadores devem ter até 32 caracteres significativos (não podendo ser idêntico ao nome das palavras reservadas). O nome de um identificador deve ser iniciado por letras ou o caracter underline (`_`). O resto é formado por letras, dígitos e caracter underline. Não é permitida a utilização de espaços para a formação do identificador e o C diferencia caracteres maiúsculos de minúsculos. Exemplo de identificadores válidos: `ValorTotal`, `valor_total`, `soma_item_1`.



Variáveis

Para que se possa usar uma variável em um programa, é necessário fazer uma declaração de variável antes. A declaração de variáveis simplesmente informa ao processador quais são os nomes utilizados para armazenar dados variáveis e quais são os tipos usados. Deste modo o processador pode alocar (reservar) o espaço necessário na memória para a manipulação destas variáveis. É possível declarar mais de uma variável ao mesmo tempo, basta separá-las por vírgulas. A sintaxe para declaração de variáveis:

```
tipo variavel_1 [, variavel_2, ...] ;
```

Onde `tipo` é o tipo de dado e `variavel_1` é o nome da variável a ser declarada. Se houver mais de uma variável, seus nomes são separados por vírgulas. Exemplos:

```
int i;
int x,y,z;
char letra;
float nota_1,nota_2,media;
double num;
unsigned short idade;
```

No exemplo acima, `i`, `x`, `y` e `z` foram declaradas variáveis inteiras. Assim elas podem armazenar valores inteiros de `-32.768` até `32.767`. Do mesmo modo `letra` foi declarada como variável caracter, podendo receber valores de `-128` até `127` ou caracteres do conjunto padrão ASCII. As variáveis `nota_1`, `nota_2` e `media` foram declaradas como ponto flutuante tipo `float` e `num` como ponto flutuante tipo `double`, podendo armazenar valores com casas decimais. Por fim, `idade` pode receber valores de `0` até `65.355`.

A declaração de variáveis é feita, em geral, **dentro** de uma função. Por exemplo, a função principal `main()`. Deste modo se diz que está se fazendo uma declaração de variáveis **locais**. Variáveis locais podem ser referenciadas apenas dentro da função dentro da qual foi declarada, neste caso a função `main()`. **Exemplo:** Observe o uso da declaração de variáveis no trecho de programa abaixo:

```
main()
{
```

```

float raio, area;      /* declaracao de variáveis */
raio = 2.5;
area = 3.14;
}

```

É possível fazer a declaração de variáveis **fora** de uma função. Neste caso diz-se que se fez a declaração de variáveis **globais**. O uso de variáveis globais é explicado adiante.

Observação importante: Em alguns compiladores, como no C++ da Microsoft, as variáveis não precisam ser declaradas no início da subrotina, isso significa que seu escopo pode ser reduzido a uma estrutura dentro de uma subrotina, que pode ser, por exemplo, um `if`, um `for`, etc.

Variáveis Strings

Uma string é um conjunto ordenado de caracteres (texto) que pode ser armazenado sob forma de um vetor. Esta estrutura de dados será vista em seções posteriores. Por enquanto, basta saber como declarar e armazenar um conjunto de caracteres em uma variável. **Sintaxe:** Para declarar uma variável para receber um conjunto de caracteres, deve-se adicionar um asterisco após o tipo de dado `char` e nomear a variável com um identificador qualquer, assim:

```
char* nome;
```

Isso indica que um número não determinado de caracteres poderá ser inserido nessa variável. Outra maneira é identificar o tamanho desejado da string. Isso se faz através da declaração de um vetor do tipo `char`, assim:

```
char nome[40];
```

Constantes

O C possui quatro tipos básicos de constantes: **inteiras**, de **ponto flutuante**, **caracteres** e **strings**. Constantes inteiras e de ponto flutuante representam números de um modo geral. Caracteres e strings representam letras e agrupamentos de letras (palavras).

Constantes inteiras

Uma constante inteira é um número de valor inteiro. De uma forma geral, constantes inteiras são seqüências de dígitos que representam números inteiros. A seguir são mostradas algumas constantes inteiras decimais válidas:

```
0    3    -45    26338    -7575    1010
```

Algumas constantes inteiras decimais inválidas:

```

1.                (ponto)
1,2              (vírgula)
045              (primeiro dígito é 0: não é constante decimal)
212-22-33        (caracter ilegal: -)

```

Constantes de ponto flutuante

Números **reais** (não inteiros) são representados em base 10, por números com um ponto decimal e (opcionalmente) um expoente. Um número ponto flutuante **deve** ter um ponto decimal que não pode ser substituído por uma vírgula. Um número de ponto flutuante pode ser escrito em notação científica. Neste caso o $\times 10$ é substituído por **e** ou **E**. O número 1.23e4 representa 1.23×10^4 ou 12300. A seguir são mostradas algumas constantes de ponto flutuante válidas:

0.234 125.65 .93 1.23e-9 -1.e2 10.6e18 -.853E+67

A forma de representação de um número real em C é bastante flexível. O número 314, por exemplo, pode ser representado por qualquer uma das seguintes formas:

314. 3.14e2 +3.14e+2 31.4e1 .314E+3 314e0

Constantes caracteres

Uma constante caracter é uma letra ou símbolo colocado entre **aspas simples** ou **apóstrofes**. A seguir estão representadas algumas constantes caracteres.

'a' 'b' 'x' '&' '{' ' '

Embora sejam **visualizados** como letras e símbolos, as constantes caracteres são armazenadas internamente pelo computador como um número **inteiro** entre 0 e 255. O caracter 'A' por exemplo, tem valor 65. Os valores numéricos dos caracteres estão padronizados em uma tabela chamada de *American Standard Code for Information Interchange Table* ou simplesmente *tabela ASCII*.

A tabela ASCII é formada por um conjunto de 256 códigos usados para representar caracteres ou instruções de controle em um computador. Podemos dividir estes códigos em três conjuntos: **controle, padrão e estendido**.

Os primeiros 32 códigos (0 a 31), formam o **conjunto de controle** ASCII. Estes códigos são usados para controlar dispositivos, por exemplo, uma impressora ou o monitor de vídeo. O código 12 (form feed) recebido por uma impressora gera um avanço de uma página. O código 13 (carriage return) é enviado pelo teclado quando a tecla ENTER é pressionada. Embora exista um padrão, alguns poucos dispositivos tratam diferentemente estes códigos e é necessário consultar o manual para saber exatamente como o equipamento lida com o código.

Os 96 códigos seguintes (32 a 127) formam o **conjunto padrão** ASCII. Todos os computadores lidam da mesma forma com estes códigos. Eles representam os caracteres usados na manipulação de textos: códigos-fonte, documentos, mensagens de correio eletrônico, etc. São constituídos das letras do alfabeto latino (minúsculo e maiúsculo) e alguns símbolos usuais.

Os 128 códigos restantes (128 a 255) formam o **conjunto estendido** ASCII. Estes códigos também representam caracteres imprimíveis, porém cada fabricante decide como e quais símbolos usar. Nesta parte do código estão definidos os caracteres especiais: é, ç, ã, ü ...

Qualquer caracter do conjunto ASCII pode ser mostrado na tela através da combinação das teclas ALT + código ASCII digitado no teclado numérico.

Dec.	Controle
0	NUL (Null)
1	SOH (Start of Heading)

2	STX (Start of Text)
3	ETX (End of Text)
4	EOT (End of Transmission)
5	ENQ (Enquiry)
6	ACK (Acknowledge)
7	BEL (Bell)
8	BS (Backspace)
9	HT (Horizontal Tab)
10	LF (Line Feed)
11	VT (Vertical Tab)
12	FF (Form Feed)
13	CR (Carriage Return)
14	SO (Shift Out)
15	SI (Shift In)
16	DLE (Data Link Escape)
17	DC1 (Device control 1)
18	DC2 (Device control 2)
19	DC3 (Device control 3)
20	DC4 (Device control 4)
21	NAK (Negative Acknowledge)
22	SYN (Synchronous Idle)
23	ETB (End Transmission Block)
24	CAN (Cancel)
25	EM (End of Media)
26	SUB (Substitute)
27	ESC (Escape)
28	FS (File Separator)
29	GS (Group Separator)
30	RS (Record Separator)
31	US (Unit Separator)

Caracter	Dec.
<espaço>	32
!	33
"	34
#	35
\$	36
%	37
&	38
'	39




(40
)	41
*	42
+	43
,	44
-	45
.	46
/	47
0	48






1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57

:	58
;	59
<	60
=	61
>	62
?	63
@	64
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74
K	75
L	76
M	77
N	78
O	79
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90
[91
\	92
]	93
^	94
_	95

`	96
a	97
b	98
c	99
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109
n	110
o	111
p	112
q	113
r	114
s	115
t	116
u	117
v	118
w	119
x	120
y	121
z	122
{	123
	124
}	125
~	126
<delete>	127
Ç	128
ü	129
é	130
â	131
ä	132
à	133

å	134
ç	135
ê	136
ë	137
è	138
ï	139
î	140
ì	141
Ä	142
Å	143
É	144
æ	145
Æ	146
ô	147
ö	148
ò	149
û	150
ù	151
ÿ	152
Ö	153
Ü	154
ç	155
£	156
¥	157
₤	158
f	159
ááááá	160
í	161
ó	162
ú	163
ñ	164
Ñ	165
ª	166
º	167
¿	168
┐	169
└	170
½	171

$\frac{1}{4}$	172
i	173
«	174
»	175
	176
	177
	178
	179
└	180
┐	181
┌	182
└	183
┐	184
┌	185
└	186
┐	187
┌	188
└	189
┐	190
┌	191
└	192
┐	193
┌	194
└	195
┐	196
┌	197
└	198
┐	199
┌	200

┐	201
┌	202
┐	203
└	204
=	205
┐	206
┌	207
┌	208
┐	209
π	210
ℓ	211
ℓ	212
ℓ	213
π	214
┐	215
┐	216
┐	217
┐	218
	219
	220
	221
	222
	223
α	224
β	225
Γ	226
π	227
Σ	228
σ	229

μ	230
τ	231
Φ	232
Θ	233
Ω	234
δ	235
∞	236
φ	237
∈	238
∩	239
≡	240
±	241
≥	242
≤	243
┌	244
┐	245
÷	246
≈	247
°	248
•	249
•	250
√	251
n	252
z	253
•	254
	255

Entre os caracteres da tabela ASCII estendidos, os mais úteis estão, talvez, os caracteres de desenho de quadro em linhas simples e duplas: os caracteres de 179 a 218. Como a visualização deste conjunto é difícil, o desenho abaixo pode auxiliar nesta tarefa:

		196	194					205	203				
218	┌	—	┐		└	191	201	▯	=	▯		└	187
179							186	▯					
195	└		┌		└	180	204	▯		▯		▯	185
			197							206			
192	┌		┐		└	217	200	▯		▯		▯	188
			193							202			
			209							210			
213	┌		┐		└	184	214	▯		▯		└	183
198	└		┌		└	181	199	▯		▯		▯	182
			216							215			
212	┌		┐		└	190	211	▯		▯		▯	189
			207							208			

Certos códigos de controle da tabela ASCII (como o *line feed*) ou caracteres especiais (como o ') possuem representação especial no C. Esta representação chama-se **seqüência de escape** representada por uma *barra invertida* (\) e um *caracter*. Seqüências de escape são interpretadas como caracteres simples. Abaixo segue uma lista das principais seqüências de escape usadas no C.

Controle/Caracter	Sequencia de escape	Valor ASCII
nulo (null)	\0	00
campainha (bell)	\a	07
retrocesso (backspace)	\b	08
tabulacao horizontal	\t	09
nova linha (new line)	\n	10
tabulacao vertical	\v	11
alimentacao de folha (form feed)	\f	12
retorno de carro (carriage return)	\r	13
aspas (")	\"	34
apostrofo (')	\'	39
interrogacao (?)	\?	63
barra invertida (\)	\\	92

Constantes strings

Uma constante string consiste de um conjunto de caracteres colocados entre **aspas duplas**. Embora as instruções do C usem apenas os caracteres do conjunto padrão ASCII, as constantes character e string podem conter caracteres do conjunto estendido ASCII: é, ã, ç, ü, ... A seguir são mostradas algumas constantes strings válidas:

```
"Oba! "  
"Porto Alegre"  
"A resposta é: "  
"João Carlos da Silveira"  
"a"  
"isto é uma string"
```

Constantes Simbólicas

Muitas vezes uma constante numérica é identificada por um símbolo: `Pi = 3.14159`, por exemplo. Pode-se definir um nome simbólico para esta constante, isto é, podemos definir uma **constante simbólica** que represente valor.

Constantes definidas pelo programador

O programador pode definir constantes simbólicas em qualquer programa. A sintaxe da instrução de definição de uma constante simbólica é:

```
#define nome valor
```

onde `#define` é uma diretiva de compilação que diz ao compilador para trocar as ocorrências do texto `nome` por `valor`. Observe que **não há ;** no final da instrução, pois trata-se de um comando para o compilador e não para o processador. A instrução `#define` **deve** ser escrita **antes** da instrução de declaração da rotina principal. Por convenção, as constantes definidas pelo programador são maiúsculas, para diferenciar das variáveis. Exemplos de constantes simbólicas:

```
#define PI 3.14159  
#define ON 1  
#define OFF 0  
#define ENDERECO 0x378  
  
void main()  
{  
...  
}
```

No exemplo acima, foi definida a constante `PI` com o valor `3.14159`. Isto significa que todas as ocorrências do texto `PI` serão trocadas por `3.14159`. Assim, se for escrita uma instrução:

```
area = PI * raio * raio;
```

O compilador vai interpretar esta instrução como se fosse escrita assim:

```
area = 3.14159 * raio * raio;
```

Poderia ser utilizada uma variável no lugar da constante, assim:

```
float pi = 3.14159;  
area = pi * raio * raio;
```

Porém, este tipo de abordagem tem duas **desvantagens**. Primeiro, reserva 4 bytes de memória desnecessariamente. Segundo, esta instrução é executada mais lentamente, pois o processador precisa acessar a memória para verificar qual é o valor de `pi`.

O uso da diretiva `#define` não se restringe apenas ao apresentado acima, podendo ser utilizada para definir **macro instruções**.

Constantes declaradas (const)

Também é possível, com o prefixo `const`, declarar constantes com um tipo específico, exatamente como faria com uma variável (claro que não é possível alterar o valor):

```
const int width = 100;  
const char tab = '\t';  
const zip = 12440;
```

Em caso do tipo não ter sido especificado (como no último exemplo) o compilador assume que é do tipo `int`.

Constantes pré-definidas

Algumas constantes simbólicas já estão pré-definidas. Estas constantes em geral definem alguns valores matemáticos, limites de tipos, etc. A seguir segue uma tabela contendo algumas (existem muitas outras) constantes simbólicas pré-definidas no compilador Turbo C da Borland.

Biblioteca	Constante	Valor	Significado
math.h	M_PI	3.14159...	π
math.h	M_PI_2	1.57079...	$\pi/2$
math.h	M_PI_4	0.78539...	$\pi/4$
math.h	M_1_PI	0.31830...	$1/\pi$
math.h	M_SQRT2	1.41421...	$\sqrt{2}$
conio.h	BLACK	0	valor da cor preta
conio.h	BLUE	1	valor da cor azul
conio.h	GREEN	2	valor da cor verde
conio.h	CYAN	3	valor da cor cyan
conio.h	RED	4	valor da cor vermelho
conio.h	MAGENTA	5	valor da cor magenta

limits.h	INT_MAX	32767	limite superior do tipo int
limits.h	INT_MIN	-32768	limite inferior do tipo int

Cada uma das constantes acima esta definida em uma biblioteca. Uma biblioteca, em C, é um arquivo pré-compilado chamado arquivo **header** (cabeçalho, em inglês). Em cada biblioteca estão agrupadas constantes e funções com características em comum. Por exemplo, constantes e funções matemáticas estão guardadas na biblioteca `math.h` (*mathematical functions*), constantes e funções de manipulação de teclado e monitor estão guardadas na biblioteca `conio.h` (*console input and output*). Para que se possa usar a constante simbólica em um programa é preciso **incluir** a biblioteca na compilação do programa, assim:

```
#include <nome_bib>
```

Onde `nome_bib` é o nome da biblioteca que se deseja incluir. Esta instrução deve ser escrita antes do programa principal. **Exemplo:** O programa abaixo usa a constante predefinida `M_PI` para calcular a área de um disco circular.

```
#include <math.h>

void main()
{
    float area, raio = 5.0;
    area = M_PI * raio * raio;
}
```

Atribuição de Valor

Uma das operações fundamentais na programação de computadores é a atribuição de valor a uma variável. No C, o operador de atribuição é o `=`, como já foi visto em exemplos anteriores. Exemplo de declaração de variáveis locais e globais e atribuição de valor:

```
#include <stdafx.h>

double var_d;      /* variáveis globais */
long var_l;

void main()
{
    int var_i1, var_i2, var_i3;      /* variáveis locais */
    float var_f = 5.6;
    char var_c = 'a';

    var_l = 50000;
    var_d = 100.5675;
    var_l = var_d;      /* var_l = 100 */
    var_i2 = var_c;      /* var_i2 = 97 */
    var_i1 = var_i2 = var_i3 = 0;
    getchar();      /* aguarda até uma tecla seja pressionada */
}
```

Todas as variáveis devem ser declaradas antes de serem utilizadas. Observa-se que é possível realizar a atribuição de valor na própria declaração da variável. O C possui um nível alto de conversão de tipos, isso significa que é possível, por exemplo, atribuir um valor caracter para uma variável inteira. Adicionalmente, não é possível realizar uma operação de atribuição à uma variável `string`, apesar de alguns compiladores aceitarem isso. Por exemplo, a atribuição mostrada a seguir não é válida em alguns compiladores. O C++ permite essa construção, desde que a variável não seja declarada como vetor de caracteres. A forma correta para vetores será vista em seções posteriores.

Essa construção é possível:

```
char* nome;  
nome = "João da Silva"; /* Incorreto em alguns compiladores */
```

Essa construção não é possível:

```
char nome[40];  
nome = "João da Silva"; /* Incorreto em alguns compiladores */
```

Porém, na declaração, é possível fazer a atribuição em ambas as formas:

```
char* nome = "João";  
char nome2[40] = "Maria";
```

Instruções de Entrada e Saída de Dados

O objetivo de se escrever programas é, em última análise, a obtenção de resultados (**saídas**) depois da elaboração de cálculos ou pesquisas (**processamento**), através do fornecimento de um conjunto de dados ou informações conhecidas (**entradas**).

Para que o programa possa receber dados e aloca-los em variáveis, que serão responsáveis por armazenar as informações iniciais, a linguagem deverá conter um conjunto de instruções que permitam ao operador interagir com o programa fornecendo os dados quando estes forem necessários.

Em C existem várias maneiras de fazer a leitura e escrita de informações. Estas operações são chamadas de operações de entrada e saída.

Instruções de Saída de Dados

Instrução: `printf()`

Biblioteca: `stdio.h`

Declaração: `int printf (const char* st_contr [, lista_arg]);`

Propósito: A função `printf()` (*print formatted*) imprime dados da lista de argumentos `lista_arg` formatados de acordo com a string de controle `st_contr`. Esta função retorna um valor inteiro representando o número de caracteres impressos.

Observação Importante: Além do `printf()`, no C++ há o `printf_s()`, que é considerada uma instrução com “reforço de segurança” no tratamento de `strings`. Isso significa que o ideal é utilizar o `printf_s()`, por ser recomendado. Isso acontece com várias outras funções, como o `scanf_s()`, `strcpy_s()`, `strcat_s()`, etc...

Exemplo:

```
void main()
{
    printf("Ola Mundo!!! ");
    printf("linha 1 \nlinha 2");
}
```

O resultado será o a seguir apresentado:

```
Ola Mundo!!!
linha 1
linha 2
```

Observe que, na primeira instrução, a saída é exatamente igual a string de controle. Já na segunda instrução a impressão se deu em duas linhas. Isto se deve ao `\n` que representa a seqüência de escape para nova linha.

É possível reservar espaço para o **valor** de alguma variável usando **especificadores de formato**. Um especificador de formato marca o **lugar** e o **formato** de impressão das variáveis contidas na **lista variáveis**. Deve haver um especificador de formato para cada variável a ser impressa. Todos os especificadores de formato começam com um `%`.

Observe, no exemplo abaixo, as instruções de saída formatada e os respectivos resultados. Admita que `idade` seja uma variável `int` contendo o valor 29 e que `tot` e `din` sejam variáveis `float` cujos valores são, respectivamente, 12.3 e 15.0.

Exemplo:

```
printf("Tenho %d anos de vida",idade);
printf("Total: %.2f \nDinheiro: %.2f \nTroco: %.2f", tot, din, din-tot);
```

Saída:

```
Tenho 29 anos de vida
Total: 12.30
Dinheiro: 15.00
Troco: 2.70
```


Depois do sinal %, seguem-se alguns modificadores, cuja sintaxe é a seguinte:

% [flag] [tamanho] [.precisão] tipo

[flag]	justificação de saída: (Opcional)
-	justificação à esquerda.
+	conversão de sinal (saída sempre com sinal: + ou -)
<espaço>	conversão de sinal (saídas negativas com sinal, positivas sem sinal)
[tamanho]	especificação de tamanho (Opcional)
n	pelo menos n dígitos serão impressos (dígitos faltantes serão completados por brancos).
0n	pelo menos n dígitos serão impressos (dígitos faltantes serão completados por zeros).
[.precisão]	especificador de precisão, dígitos a direita do ponto decimal. (Opcional)
(nada)	padrão: 6 dígitos para reais.
.0	nenhum dígito decimal.
.n	são impressos n dígitos decimais.

Tipo	caracter de conversão de tipo (Requerido)
d	inteiro decimal
u	inteiro decimal sem sinal
ld	inteiro decimal longo
o	inteiro octal
x	inteiro hexadecimal
f	ponto flutuante: [-] dddd.ddd.
e	ponto flutuante com expoente: [-] d.dddde[+/-] ddd
c	caracter simples
s	string

Programa Exemplo: O programa a seguir ilustra o uso da função `printf()` usando várias combinações de *strings de controle* e *especificadores de formato*.

```
#include <stdafx.h>

void main()
{
    unsigned int var_ui=65000;
```

```

long var_l=35000;
float var_f=30.5;
int nro_caracteres=2670;

printf("Imprimindo um float %f\n",var_f); /* 30.500000 */
printf("%10.2f\n",var_f);                /* -----30.50 */
printf("%-10.2f\n",var_f);                /* 30.50 */
printf("%010.2f inteiro longo=%ld \n",var_f,var_l);
/* 0000030.50 inteiro longo=35000 */

printf("%5.7s\n","FACULDADE"); /* FACULDA */
printf("%5s\n","FACULDADE"); /* FACULDADE */
printf("%20s\n","FACULDADE"); /* -----FACULDADE */
printf("%-20s% \n","FACULDADE"); /* FACULDADE-----% */
printf("%c %d %o %x \n",'a','a','a','a'); /* a 97 141 61 */
printf("decimal sem sinal %u\n",var_ui); /* decimal sem sinal 65000 */
printf("%d\n",nro_caracteres); /* 2670 */
printf("%d %c\n",65,65); /* 65 A */
getchar();
}

```

Instrução: gotoxy() – Somente para compiladores mais antigos (MS-DOS), não existe no Visual C++

Biblioteca: `conio.h`

Declaração: `void gotoxy(int pos_x, int pos_y);`

Propósito: Em modo texto padrão, a tela é dividida em uma janela de 80 colunas e 25 linhas. A função `gotoxy()` permite o posicionamento do cursor em qualquer posição (`pos_x,pos_y`) da tela, sendo que a posição (1,1) corresponde ao canto superior esquerdo da tela e a posição (80,25) corresponde ao canto inferior direito. Normalmente é utilizado para posicionar o cursor antes de mostrar algo na tela.

Exemplo:

```

void main()
{
    gotoxy(34,12);
    printf("Ola Mundo!!! ");
    getchar();
}

```

O resultado será a mostragem da string "Ola Mundo!!! " no meio da tela.

Instrução: clrscr(), cleveland()– Somente para compiladores mais antigos (MS-DOS), não existe no Visual C++

Biblioteca: `conio.h`

Declaração: `void clrscr(void);`
`void clreol(void);`

Propósito: A função `clrscr()` (*clear screen*) limpa a janela de tela e posiciona o cursor na primeira linha e primeira coluna da janela (canto superior esquerdo da janela). A função `clreol()` (*clear to end of line*) limpa uma linha desde a posição do cursor até o final da linha mas não modifica a posição do cursor.

Exemplo:

```
void main()
{
    clrscr();
    gotoxy(30,10);
    printf("Ola Mundo!!! ");
    getchar();
    gotoxy(33,10);
    clreol();
    getchar();
}
```

O resultado será a mostragem da string "Ola Mundo!!! " após limpar a tela. Ao ser teclado algo, o que permanecerá na tela será apenas "Ola", com o cursor na posição seguinte.

Instrução: putchar()

Biblioteca: `stdio.h`

Declaração: `int putchar(int c);`

Propósito: A função `putchar()` (*put character*) imprime um caracter individual `c` na saída padrão (em geral o monitor de vídeo), a partir da posição do cursor. O caracter deve estar entre apóstrofes ou aspas simples.

Exemplo:

```
void main()
{
    printf("Ola Mundo");
    putchar('!');
    putchar('!');
    putchar('!');
    getchar();
}
```

O resultado será a mostragem da string "Ola Mundo!!!".

Instrução: puts()

Biblioteca: `stdio.h`

Declaração: `int puts(const char *s);`

Propósito: A função `puts()` (*put string*) imprime um string `s` na saída padrão (em geral o monitor de vídeo), a partir da posição do cursor. É semelhante ao `printf()`, mas não permite mostrar valores de variáveis formatadas. Além disso, o `puts()` dá um salto de linha depois de mostrar a string, o que o `printf()` não faz.

Exemplo:

```
void main()
{
    puts("Ola Mundo!!! ");
    getchar();
}
```

O resultado será a mostragem da string "Ola Mundo!!! ".

Instruções de Entrada de Dados

Instrução: scanf()

Biblioteca: `stdio.h`

Declaração: `int scanf(const char* st_contr, end_var);`

Propósito: A função `scanf()` (*scan formatted*) permite a entrada de dados numéricos, caracteres e strings e sua respectiva atribuição a variáveis cujos endereços são `end_var`. Esta função é dita de entrada formatada, pois os dados de entrada são formatados pela string de controle `st_contr` a um determinado tipo de variável (`int`, `float`, `char`, ...). O uso da função `scanf()` é semelhante ao da função `printf()`. A função lê da entrada padrão (em geral, teclado) uma lista de valores que serão formatados pela string de controle e armazenados nos endereços das variáveis da lista. A string de controle é formada pelo mesmo conjunto de especificadores de formato do `printf()`.

A lista de variáveis é o conjunto de (endereços de) variáveis para os quais serão passados os dados lidos. Variáveis simples devem ser precedidos pelo caracter `&`. Variáveis string e vetores não são precedidos pelo caracter `&`.

Sempre que se utiliza o `scanf()`, é necessária a utilização do `fflush(stdin);`, evitando que “sujeira” fique para a próxima leitura. É necessário se ter o `#include <stdio.h>` para o `fflush` funcionar.

Observação Importante: Como explicado no `printf()`, é recomendada a utilização do `scanf_s()` no lugar do `scanf()`.

Exemplo:

```
#include <stdafx.h>

void main()
{
    int idade;
    char nome[30];

    printf("Digite sua idade: ");
    scanf("%d",&idade);
    fflush(stdin); /* se não utilizar o fflush, pode-se ter problemas */
    printf("Digite seu nome: ");
    scanf("%s",nome); /* Strings não utilizar '&' na leitura */
    fflush(stdin); /* se não utilizar o fflush, pode-se ter problemas */
    printf("%s, você tem %d anos. \n", nome, idade);
    getchar();
}
```

Instrução: `getchar()`, `getch()`, `getche()`

Biblioteca: `conio.h`

Declaração: `int getchar(void);`
 `int getch(void);` // necessita o `#include <conio.h>`
 `int getche(void);` // necessita o `#include <conio.h>`

Propósito: Estas funções fazem a leitura de um único caracter do teclado. A diferença entre o `getch()` (*get character*) e o `getche()` (*get character and echoe*) é que o segundo mostra o caracter digitado na tela e o primeiro não. O `getche()` é semelhante ao `scanf()` com a utilização do `%c`, porém, alguns problemas são detectados ao se utilizar o `scanf()`, por isso o `getche()` é o mais utilizado para a leitura de um único caracter. O `getchar()` tem a mesma função que o `getche()`, porém necessita do `fflush(stdin);` para limpar a sujeira do buffer de teclado.

Exemplo:

```
#include <stdafx.h>
#include <conio.h>

void main()
{
    char character;

    printf("Digite um caracter qualquer: ");
    character=getch();
    printf("\nO caracter digitado foi %c",character);
}
```

```
printf("\nDigite outro caracter: ");
caracter=getche();
printf("\nO caracter digitado foi %c",caracter);
getchar();
}
```

Instrução: gets()

Biblioteca: `stdio.h`

Declaração: `int *gets(char *string);`

Propósito: A função `gets()` (*get string*) faz a leitura de um conjunto de caracteres (string) e armazena em uma variável string. É semelhante ao `scanf()` com a utilização do `%s`, porém o ideal é sempre utilizar o `gets()` para ler strings.

Exemplo:

```
#include <stdafx.h>

void main()
{
    char nome[30];

    printf("Digite seu nome: ");
    gets(nome);
    printf("\nO seu nome é %s. ",nome);
    getchar();
}
```

Dicas Válidas

Para facilitar a leitura de valores, pode-se adotar as normas:

1. Para ler variáveis inteiras (`%d`) e de ponto flutuante (`%f`), utilizar sempre o `scanf` com o `fflush`, não esquecendo do `#include <stdio.h>` Exemplo:

```
printf("Digite valor inteiro: ");
scanf("%d",&valorinteiro);
fflush(stdin);
printf("Digite valor real: ");
scanf("%f",&valorflutuante);
fflush(stdin);
```

2. Para ler variáveis caracter, utilizar sempre o `getche()`, sem se esquecer do `#include <conio.h>`. Exemplo:

```
printf("Digite caracter: ");
```

```
character=getche();
```

Ou usar o `getchar()`, juntamente com o `fflush`. Exemplo:

```
printf("Digite character: ");
character=getchar();
fflush(stdin);
```

3. Para ler variáveis string, utilizar sempre o `gets()`. Exemplo:

```
printf("Digite nome: ");
gets(nome);
```

Exercícios de Entrada e Saída de Dados

1. Fazer um programa que leia os dados de um cliente qualquer, composto por número de identificação, nome completo, endereço, cidade, UF, telefone, cpf, sexo, idade e limite de crédito. Após, mostrar todos os dados, um em cada linha, utilizando um único `printf`. O limite de crédito deve ser mostrado com duas casas decimais.
2. Fazer um programa que leia uma variável inteira e após mostre-a como sendo um character. Faça também a leitura de uma variável character e mostre-a como inteiro. Faça outros testes declarando um tipo de dado e mostrando com outro especificador de formato.

Expressões e Operadores e Funções Matemáticas

Um programa tem como característica fundamental a capacidade de processar dados. Processar dados significa realizar operações com estes dados. As operações a serem realizadas com os dados podem ser determinadas por **operadores** ou **funções**. Os operadores podem ser de **atribuição**, **aritméticos**, **de atribuição aritmética**, **incrementais**, **relacionais**, **lógicos** e **condicionais**. **Exemplo:** o símbolo `+` é um *operador* que representa a operação aritmética de adição. O identificador `sqrt()` é uma *função* que representa a operação de extrair a raiz quadrada de um número.

Uma **expressão** é um arranjo de operadores e operandos. A cada expressão válida é atribuído um valor numérico. **Exemplo:** `4 + 6` é uma expressão cujo valor resultante é `10`. A expressão `sqrt(9.0)` tem como valor resultante `3.0`.

Operador de Atribuição

A operação de atribuição é a operação mais simples do C. Consiste de atribuir valor de uma **expressão** a uma **variável**, como já foi visto em seções anteriores. **Sintaxe:** A sintaxe da operação de atribuição é a seguinte:

```
identificador = expressão;
```

Onde `identificador` é o nome de uma variável e `expressão` é uma expressão válida (ou outro identificador). **Exemplo:** A seguir são mostradas algumas atribuições válidas:

```
a = 1;
delta = b * b - 4.3 * a * c;
i = j;
```

É bom observar que o operando esquerdo **deve ser** um identificador de variável, isto é, **não pode** ser uma constante ou expressão. **Exemplo:** A seguir são mostradas algumas atribuições inválidas:

```
1 = a;
b + 1 = a;
```

Conversão de Tipo

Se os dois operandos de uma atribuição **não são** do mesmo tipo, o valor da expressão ou operador da direita **será convertido** para o tipo do identificador da esquerda (não são todos os casos). **Exemplo:** Algumas atribuições com conversão de tipo:

```
int i;
float r;
i = 5;          /* valor de i: 5 */
r = i;          /* valor de r: 5.0 */
```

Nestas conversões podem ocorrer alterações dos valores convertidos se o operando da esquerda for de um tipo que utilize **menor número** de bytes que o operando da direita. **Exemplo:** Algumas atribuições com conversão de tipo e perda de informação:

```
int i;
float r = 654.321;
i = r;          /* valor de i: 654 (truncamento) */
```

Pode-se dizer que as conversões potencialmente perigosas (onde há possibilidade de perda de informação) são:

```
char ← int ← float ← double
```

Observe que o compilador C, ao encontrar estas situações, **não gera** nenhum aviso de atenção para o programador. Assim, este detalhe pode gerar um erro lógico de programação que passe despercebido ao programador inexperiente. É possível dizer que a linguagem C possui tipos “macios” (*soft types*), pois a operação com variáveis de tipos diferentes é perfeitamente possível. Esta característica do C se contrapõe a algumas linguagens em que isto não é possível. Estas linguagens possuem tipos “duros” (*hard types*).

Limites de Intervalo do Tipo de Dado

Os tipos em C tem intervalos bem definidos e os resultados das operações devem respeitar estes intervalos. Se for atribuído a uma variável um valor que esteja fora dos seus limites então haverá perda de informação. **Exemplo:** Observe as expressões abaixo, assumindo que `i` seja uma variável do tipo `short`.

```
i = 4999;           /* o valor de i e'  4999 */
i = 4999 + 1;       /* o valor de i e'  5000 */
i = 5000 + 30000;   /* o valor de i e' -30536 */
```

O valor `35000` ultrapassou o limite superior do tipo `short` (`32767`). É importante observar que em C, ao contrário de outras linguagens, a ultrapassagem do limite de um tipo **não é interpretado como erro**. Isto pode acarretar resultados inesperados para o programador desatento.

Atribuição Múltipla

É possível atribuir um valor a muitas variáveis em uma única instrução. A esta operação dá-se o nome de **atribuição múltipla**. **Sintaxe:**

```
var_1 = [var_2 = ... ] expressão;
```

Na atribuição múltipla as operações ocorrem da **direita** para a **esquerda**, isto é, inicialmente o valor de `expressão` é atribuído a `var_2` e depois o valor de `var_2` é atribuído a `var_1`. Deve-se tomar cuidado com as conversões de tipo e limites de intervalo para atribuições de tipos diferentes. **Exemplo:** Observe a instrução de atribuição múltipla a seguir: as variáveis inteiras `i`, `j` e `k` são todas inicializadas com o valor `1`. E as variáveis de dupla precisão `max` e `min` são inicializadas com o valor `0.0`:

```
int i, j, k;
double max, min;
i = j = k = 1;
max = min = 0.0;
```

Exercícios de Atribuição de Valor

1. Fazer um programa que declare quatro variáveis dos tipos (`unsigned short`, `short`, `unsigned long` e `long`). Após, faça a atribuição a essas variáveis dos limites máximos de cada tipo `+1` e mostre para ver o resultado.
2. Fazer um programa que leia uma variável `float x` e uma variável `int y`. Após, efetuar a atribuição da variável `x` a uma variável `long z` e a atribuição da variável `y` a uma variável `char w`. Mostre as variáveis `z` e `w` para ver o resultado.

Operadores Aritméticos

Existem cinco operadores aritméticos em C. Cada operador aritmético está relacionado ao uma operação aritmética elementar: adição, subtração, multiplicação e divisão. Existe ainda um operador (%) chamado operador de **módulo**, cujo significado é o resto da divisão inteira. Os símbolos dos operadores aritméticos são:

Operador	Operação
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo (resto da divisão inteira)

Sintaxe: A sintaxe de uma expressão aritmética é:

`operando operador operando`

Onde `operador` é um dos **caracteres** mostrados acima e `operando` é uma **constante** ou um identificador de **variável**. **Exemplo:** Algumas expressões aritméticas:

`1+2 a-4.0 b*c valor_1/taxa num%2`

Não existe em C, como existe em outras linguagens, um operador específico para a operação de potenciação (a^b). Existe, porém, uma função de biblioteca (`pow()`) que realiza esta operação. Embora as operações do C sejam semelhantes às operações aritméticas usuais da matemática, alguns detalhes são específicos da linguagem, e devem ser observados.

Restrições de Operandos

Os operandos dos operadores aritméticos devem ser constantes numéricas ou identificadores de variáveis numéricas. Os operadores `+`, `-`, `*`, `/` podem operar números de todos os tipos (inteiros ou reais), porém o operador `%` somente aceita operandos **inteiros**.

Exemplo: Expressões válidas:

Expressão	Resultado
<code>6.4 + 2.1</code>	<code>8.5</code>
<code>7 - 2</code>	<code>5</code>
<code>2.0 * 2.0</code>	<code>4.0</code>
<code>6 / 3</code>	<code>2</code>
<code>47 % 2</code>	<code>1</code>

A expressão a seguir é inválida, pois o primeiro operando não é um numero inteiro.

`6.4 % 3 /* invalido! */`

Uma restrição ao operador de divisão (/) é que o denominador **deve** ser diferente de zero. Se alguma operação de divisão por zero for realizada ocorrerá um **erro de execução (lançamento de exceção)** do programa (*run-time error*), o programa será abortado e a mensagem `divide error` (ou `division by zero`) será exibida.

Casting (Type Cast ou Conversão de Tipo)

Pode-se contornar o problema do operador inteiro da operação módulo usando o artifício da conversão de tipo (*casting*). Algumas vezes é necessário, momentaneamente, modificar o tipo de dado representado por uma variável, isto é, se quer que o dado seja apresentado em um tipo diferente do qual a variável foi inicialmente declarada. Por exemplo: declara-se uma variável como `int` e se quer, momentaneamente, que seu conteúdo seja apresentado como `float`. Este procedimento é chamado de **conversão de tipo** ou *casting*. **Sintaxe:** A sintaxe da instrução de conversão de tipo é:

```
(tipo) variável
```

Exemplo: Observe o trecho de programa a seguir:

```
int num;
float valor = 13.0;
num = valor % 2;      /* inválido! */
num = (int)valor % 2; /* válido! */
```

Precedência de Operadores

Quando mais de um operador se encontram em uma expressão aritmética, as operações são efetuadas uma de cada vez, respeitando algumas regras de precedência: Estas regras de precedência são as mesmas da matemática elementar.

Os operadores de multiplicação (*), divisão (/) e módulo (%) tem precedência sobre os operadores de adição (+) e subtração (-). Entre operadores de mesma precedência as operações são efetuadas da **esquerda** para a **direita**. **Exemplo:** Observe, nas expressões abaixo, o seu valor e a ordem das operações efetuadas:

Expressão	Valor	Ordem
<code>1 + 2 - 3</code>	0	+ -
<code>24 - 3 * 5</code>	9	* -
<code>4 - 2 * 6 / 4 + 1</code>	2	* / - +
<code>6 / 2 + 11 % 3 * 4</code>	11	/ % * +

A ordem de precedência dos operadores pode ser quebrada usando-se parênteses: (). Os parênteses são, na verdade, operadores de mais alta precedência e são executados primeiro. Parênteses internos são executados primeiro que parênteses externos. **Exemplo:** Observe, nas expressões abaixo, o seu valor e a ordem das operações efetuadas:

Expressão	Valor	Ordem
<code>1 + (2 - 3)</code>	0	- +

$(24 - 3) * 5$	105	$- *$
$(4 - 2 * 6) / 4 + 1$	-1	$* - / +$
$6 / ((2 + 11) \% 3) * 4$	24	$+ \% / *$

Observe que os operadores e os operandos deste exemplo são os mesmos do exemplo anterior. Os valores, porém, são diferentes, pois a ordem de execução das operações foi modificada pelo uso dos parênteses.

Exercícios com Operadores Aritméticos

1. Fazer um programa que faça a leitura do número de um funcionário, seu número de horas trabalhadas, o valor que recebe por hora, o seu sexo (**M** ou **F**) e calcula o salário mostrando com a mensagem: "O funcionário número xxx, do sexo X possui salário xxx".
2. Fazer um programa que leia **p**, **u** e **r**, respectivamente o 1º termo de uma progressão aritmética, o último termo da progressão e a razão desta progressão. Determinar a soma dos termos desta progressão aritmética e mostrar.

$$n = \frac{a_n - a_1 + r}{r} \quad S_n = \frac{(a_1 + a_n) \cdot n}{2}$$

3. Criar um programa que leia um valor em uma variável **int**. Calcular e mostrar o menor número possível de notas de 100, 50, 20, 10, 5, 2 e 1 em que o valor pode ser decomposto. Ex.: 389 pode ser decomposto em 3 notas de 100, 1 de 50, 1 de 20, 1 de 10, 1 de 5, 2 de 2 e 0 de 1. Dica: Utilizar o resto da divisão de inteiros (%).

Operadores de Atribuição Aritmética

É comum em programação ter que alterar o valor de uma variável realizando alguma operação aritmética com ela, como **a = a + 1** ou **valor = valor * 5**. Embora seja perfeitamente possível escrever estas instruções, foi desenvolvido na linguagem C uma forma **otimizada**, através do uso de operadores ditos **operadores de atribuição aritmética**, que são **+=**, **-=**, ***=**, **/=**, **%=**. Deste modo, as instruções acima podem ser reescritas assim: **a += 1** e **valor *= 5**, respectivamente. A sintaxe da atribuição aritmética é a seguinte:

```
var += exp;
var -= exp;
var *= exp;
var /= exp;
var %= exp;
```

Onde **var** é o identificador da variável e **exp** é uma expressão válida. Estas instruções são equivalentes (mas não iguais) às seguintes:

```
var = var + exp;
var = var - exp;
```

```
var = var * exp;
var = var / exp;
var = var % exp;
```

Exemplo: Observe as atribuições aritméticas abaixo e suas instruções equivalentes:

Atribuição aritmética	Instrução equivalente
<code>i += 1;</code>	<code>i = i + 1;</code>
<code>j -= val;</code>	<code>j = j - val;</code>
<code>num *= 1 + k;</code>	<code>num = num * (1 + k);</code>
<code>troco /= 10;</code>	<code>troco = troco / 10;</code>
<code>resto %= 2;</code>	<code>resto = resto % 2;</code>

Operadores Incrementais

Na linguagem C existem instruções muito comuns chamadas de **incremento** e **decremento**. Uma operação de incremento **adiciona** uma unidade ao conteúdo de uma variável. Uma operação de decremento **subtrai** uma unidade do conteúdo de uma variável. Isso é realizado através de operadores específicos para realizar as operações de incremento (**++**) e decremento (**--**). Eles são genericamente chamados de **operadores incrementais**. **Sintaxe:**

Instrução	Instrução equivalente
<code>++var</code>	<code>var = var + 1</code>
<code>var++</code>	<code>var = var + 1</code>
<code>--var</code>	<code>var = var - 1</code>
<code>var--</code>	<code>var = var - 1</code>

Observe que existem duas sintaxes possíveis para os operadores: pode-se colocar o operador **à esquerda** ou **à direita** da variável. Nos dois casos o valor da variável será incrementado (ou decrementado) de uma unidade. Porém, se o operador for colocado **à esquerda** da variável, o valor da variável será incrementado (ou decrementado) **antes** que a variável seja usada em alguma outra operação. Caso o operador seja colocado **à direita** da variável, o valor da variável será incrementado (ou decrementado) **depois** que a variável for usada em alguma outra operação. **Exemplo:** Observe o fragmento de código abaixo e note o valor que as variáveis recebem **após** a execução da instrução:

```
int a, b, c, i = 3;      /* a: ?   b: ?   c: ?   i: 3 */
a = i++;                /* a: 3   b: ?   c: ?   i: 4 */
b = ++i;                /* a: 3   b: 5   c: ?   i: 5 */
c = --i;                /* a: 3   b: 5   c: 4   i: 4 */
```

Os operadores incrementais são bastante usados para o controle de laços de repetição, que serão vistos em seções posteriores. É importante que se conheça exatamente o efeito sutil da colocação do operador, pois isto pode enganar o programador inexperiente.

Exemplo: O programa a seguir mostra um exemplo de utilização de operadores aritméticos:

```
#include <stdafx.h>
```

```

void main()
{
    int divisor=5,dividendo=20,numero=1;
    float divisorf=5,dividendof=20;

    printf("resto= %d\n",dividendo%divisor);
    printf("resto= %d\n", (int)dividendof%(int)divisorf);
    numero=numero+'A';
    printf("inteiro=%d ascii=%c\n",numero,numero);
    divisor=numero++;
    dividendo=++numero;
    printf("divisor=%d dividendo=%d\n",divisor,dividendo);
    getchar();
}

```

Percebe-se que as expressões aritméticas podem tanto ser realizadas na atribuição a uma variável como na mostragem do seu resultado. Isso é perfeitamente possível em C.

Exercício

1. Fazer um programa que calcule o consumo de combustível de um automóvel, com os seguintes itens:

- Pedir o nome do automóvel e o seu combustível (álcool, gasolina ou diesel);
- De acordo com o combustível informado, pedir o seu preço por litro, assim:
`Forneça o preço por litro de <combustível>;`
- Pedir a quilometragem que o hodômetro do automóvel estava marcando quando foi abastecido (tanque cheio) a primeira vez (início);
- Pedir a quilometragem que o hodômetro do automóvel estava marcando quando foi abastecido (tanque cheio) a segunda vez (fim);
- Pedir quantos litros foram colocados no segundo abastecimento;

Modelo de saída para a tela:

```

O automóvel VECTRA rodou 99.999 Km.
Usou 999 litros de <combustível>
Fez uma média de 99,999 Km/l.
O custo foi de R$ 999,99.

```

2. O volume de uma esfera de raio R é $V = \frac{4}{3}\pi R^3$. Faça um programa que leia um raio R e imprima o volume da esfera correspondente. Utilizar a constante predefinida `M_PI` definida em `math.h` para efetuar o cálculo.

Operadores Relacionais e Lógicos

A chave para a flexibilidade de um programa é a tomada de decisões através da avaliação de condições de controle. Uma condição de controle é uma **expressão lógica** que é avaliada

como **verdadeira** ou **falsa**. Uma expressão lógica é construída com **operadores relacionais** e **lógicos**.

Operadores Relacionais

Operadores relacionais verificam a relação de magnitude e igualdade entre dois valores. São seis os operadores relacionais em C:

Operador	Significado
>	maior que
<	menor que
>=	maior ou igual a (não menor que)
<=	menor ou igual a (não maior que)
==	igual a
!=	não igual a (diferente de)

Sintaxe: A sintaxe das expressões lógicas é:

```
expressão_1 operador expressão_2
```

Onde `expressão_1` e `expressão_2` são duas expressões numéricas quaisquer, e `operador` é um dos operadores relacionais.

Importante: Em alguns compiladores C **não existem** tipos de dados lógicos (`bool`), e o **resultado** de uma expressão lógica é um **valor numérico**: uma expressão avaliada **verdadeira** recebe o valor `1`, uma expressão lógica avaliada **falsa** recebe o valor `0`. Ao inserir tipos lógicos em compiladores C, foi mantida essa mesma compatibilidade. Portanto, as expressões abaixo são idênticas:

```
bool t=true;
bool t=1;    // true

bool t=false;
bool t=0;    // false
```

Exemplo: Observe as expressões lógicas a seguir e verifique o resultado de sua avaliação. Admita que `i` e `j` são variáveis `int` cujos valores são `5` e `-3`, respectivamente. As variáveis `r` e `s` são `float` com valores `7.3` e `1.7`, respectivamente.

Expressão	Valor
<code>i == 7</code>	<code>false(0)</code>
<code>r != s</code>	<code>true(1)</code>
<code>i > r</code>	<code>false(0)</code>
<code>6 >= i</code>	<code>true(1)</code>
<code>i < j</code>	<code>false(0)</code>
<code>s <= 5.9</code>	<code>true(1)</code>

Os operadores relacionais de igualdade (`==` e `!=`) tem precedência **menor** que os de magnitude (`>`, `<`, `>=` e `<=`). Estes, por sua vez, têm precedência **menor** que os operadores

aritméticos. Operadores relacionais de mesma precedência são avaliados da esquerda para a direita.

Exemplo: Observe as expressões lógicas abaixo e verifique o resultado de sua avaliação. Admita que `m` e `n` são variáveis tipo `int` com valores 4 e 1, respectivamente.

Expressão	Valor	Ordem de Operação
<code>m + n == 5</code>	<code>true(1)</code>	<code>+</code> <code>==</code>
<code>m != 2 * n > m</code>	<code>true(1)</code>	<code>*</code> <code>></code> <code>!=</code>
<code>6 >= n < 3 - m</code>	<code>false(0)</code>	<code>-</code> <code>>=</code> <code><</code>
<code>m == n <= m > m</code>	<code>false(0)</code>	<code><=</code> <code>></code> <code>!=</code>

Operadores Lógicos

São três os operadores lógicos em C: `&&`, `||` e `!`. Estes operadores são equivalentes aos operadores lógicos `AND`, `OR` e `NOT` utilizados em muitas linguagens de programação.

Sintaxe: A sintaxe de uso dos operadores lógicos:

```

expr_1 && expr_2
expr_1 || expr_2
!expr

```

Onde `expr_1`, `expr_2` e `expr` são expressões lógicas quaisquer.

Os operadores lógicos atuam sobre expressões de quaisquer valores. Para estes operadores todo valor numérico diferente de `0(false)` é considerado `1(true)` e a negação de `0(false)` é `1(true)` e de qualquer outro valor é `0(false)`.

O resultado da operação lógica `&&` será `1(true)` somente se os dois operandos forem `1(true)`, caso contrário o resultado é `0(false)`. O resultado da operação lógica `||` será `0(false)` somente se os dois operandos forem `0(false)`, caso contrário o resultado é `1(true)`. O resultado da operação lógica `!` será `0(false)` se o operando for `1(true)` ou qualquer outro valor, e `1(true)` se o operando for `0(false)`. Abaixo mostra-se o resultado das possíveis combinações entre os operandos para cada operador lógico:

Expressão	op_1	op_2	Res
<code>op_1 && op_2</code>	<code>true(1)</code>	<code>true(1)</code>	<code>true(1)</code>
	<code>true(1)</code>	<code>false(0)</code>	<code>false(0)</code>
	<code>false(0)</code>	<code>true(1)</code>	<code>false(0)</code>
	<code>false(0)</code>	<code>false(0)</code>	<code>false(0)</code>

Expressão	op_1	op_2	Res
<code>op_1 op_2</code>	<code>true(1)</code>	<code>true(1)</code>	<code>true(1)</code>
	<code>true(1)</code>	<code>false(0)</code>	<code>true(1)</code>
	<code>false(0)</code>	<code>true(1)</code>	<code>true(1)</code>
	<code>false(0)</code>	<code>false(0)</code>	<code>false(0)</code>

Expressão	op	Res
! op	true(1)	false(0)
	false(0)	true(1)

O Operador `&&` tem precedência sobre o operador `||`. Estes dois têm precedência menor que os operadores relacionais. O operador `!` tem a mesma precedência que os operadores incrementais.

Exemplo: Observe as expressões lógicas abaixo e verifique o resultado de sua avaliação. Admita que `a`, `b` e `c` são variáveis tipo `int` com valores 0, 1 e 2, respectivamente.

Expressão	Valor	Ordem de Operação
<code>a && b</code>	false(0)	<code>&&</code>
<code>c > b a < c</code>	true(1)	<code>> < </code>
<code>a + b && !c - b</code>	true(1)	<code>! + - &&</code>
<code>!b && c a</code>	false(0)	<code>! && </code>

Operador Condicional

O operador condicional `?:` é usado em expressões condicionais. Uma expressão condicional pode ter dois valores diferentes dependendo de uma condição de controle.

Sintaxe: A sintaxe de uma expressão condicional é:

`condição ? expressão_1 : expressão_2`

onde `expressão_1` e `expressão_2` são duas expressões quaisquer, e `condição` é uma expressão lógica que será avaliada primeiro. Se o valor de `condição` for `true(1)`, então a expressão condicional assumirá o valor de `expressão_1`. Caso contrário assumirá o valor de `expressão_2`. Uma expressão condicional é equivalente a uma estrutura de decisão simples:

```
se condição então
    expressao_1
senão
    expressao_2
```

Exemplo: Observe as expressões condicionais abaixo e verifique o resultado de sua avaliação. Admita que `i`, `j` e `k` são variáveis tipo `int` com valores 1, 2 e 3, respectivamente.

Expressão	Valor
<code>i ? j : k</code>	2
<code>j > i ? ++k : --k</code>	4
<code>k == i && k != j ? i + j : i - j</code>	-1
<code>i > k ? i : k</code>	3

O operador condicional tem baixa precedência, precedendo, apenas, aos operadores de atribuição.

Exercícios com Operador Condicional

1. Considerando que um ano é bissexto a cada quatro anos, fazer um programa que leia o ano e mostre na tela se ele é bissexto ou não. Utilizar o operador condicional para realizar essa tarefa.
2. Referente ao exercício anterior, na verdade, um ano é bissexto a cada quatro anos, excetuando quando o ano for múltiplo de 100, mas não for múltiplo de 400. Então, refaça o exercício, incluindo essas restrições, continuando a utilizar o operador condicional.
3. Fazer um programa que leia duas variáveis inteiras quaisquer e, através do operador condicional descubra qual é o maior, mostrando a seguir com a mensagem: "O maior é o número xxx".

Funções de biblioteca

Uma função é um **sub-programa** (também chamada de rotina). Esta função *recebe* informações, *as processa* e *retorna* outra informação. Por exemplo, podemos ter uma função que receba um valor numérico, calcule seu logaritmo decimal e retorne o valor obtido. Existem dois tipos de funções: *funções de biblioteca* e *funções de usuário*. Funções de biblioteca são funções escritas pelos fabricantes do compilador e já estão pré-compiladas, isto é, já estão escritas em código de máquina. Funções de usuário são funções escritas pelo programador. Nesta seção trataremos somente das funções de biblioteca, funções de usuário serão vistas em seções posteriores.

O Uso de Funções

Antes de usar uma função é preciso saber como a função está declarada, isto é, quais são os parâmetros que a função recebe e qual é o seu tipo de retorno. Estas informações, no caso do Visual C++ da Microsoft, estão contidas na própria ajuda da IDE, e também ativando o auxílio de complemento de código, a partir da combinação de teclas `Ctrl+BarraEspaço`.

Sintaxe: A sintaxe de declaração de uma função é:

```
tipo_ret nome(tipo_1, tipo_2, ...)
```

onde `nome` é o nome da função, `tipo_1`, `tipo_2`, ... são os tipos (e quantidade) de parâmetros de entrada da função e `tipo_ret` é o tipo de dado de retorno da função. Além dos tipos usuais já vistos, existe ainda o tipo `void` (vazio, em inglês) que significa que aquele parâmetro é inexistente.

Exemplo: A função para descobrir o co-seno `cos()` da biblioteca `math.h` é declarada como:

```
double cos(double);
```

Isto significa que a função tem um parâmetro de entrada e um retorno, ambos do tipo `double`. A utilização dessa função deve ser parecida com o que segue:

```
x = cos(0.01);
```

Sendo que a variável `x` deve ser declarada como `double` ou `float`.

Para poder usar uma função de biblioteca deve-se **incluir** a biblioteca na compilação do programa. Esta inclusão é feita com o uso da diretiva `#include` colocada antes do `main()`, como já foi visto em seções anteriores.

Exemplo: Assim podemos usar a função no seguinte trecho de programa:

```
#include <stdafx.h>           // inclusão
#include <math.h>             // das bibliotecas

void main()
{
    double h = 5.0;           /* hipotenusa */
    double co;                /* cateto oposto */
    double alfa = 0.05;       /* angulo */
    co = h * cos(alfa);        /* calculo: uso da funcao cos() */
    printf("Resultado: %.2f", co);
    getchar();
}
```

Algumas Funções Interessantes

A seguir são listadas algumas funções disponíveis nas bibliotecas C. Dependendo da situação, elas podem ser muito úteis.

Biblioteca `math.h`

Cálculo do valor absoluto do inteiro `i` e do real `d`, respectivamente:

```
int abs(int i);
double fabs(double d);
```

Funções trigonométricas do ângulo `arco`, em radianos:

```
double sin(double arco);
double cos(double arco);
double tan(double arco);
double asin(double arco);
double acos(double arco);
double atan(double arco);
```

Funções logarítmicas: `log()` é logaritmo natural (base e), `log10()` é logaritmo decimal (base 10):

```
double log(double num);
```

```
double log10(double num);
```

Potenciação: `pow(3.2,5.6)` = $3.2^{5.6}$:

```
double pow(double base, double exp);
```

Raiz quadrada: `sqrt(9.0)` = 3.0:

```
double sqrt(double num);
```

Biblioteca `stdlib.h`

O `rand()` gera um número inteiro aleatório entre 0 e `RAND_MAX` (constante definida no `stdlib.h`, que corresponde ao valor 32767). É necessário inicializar o gerador de números aleatórios antes, através do `srand()`. Normalmente, a chamada semente de inicialização (seed) é a hora do sistema (`time`), que precisa da inclusão da biblioteca `time.h`.

```
void srand(int semente);  
int rand(void);
```

Exemplo:

```
#include <stdafx.h>  
#include <stdlib.h>  
#include <time.h>  
  
void main()  
{  
    int valor;  
  
    // Inicializa o gerador de números aleatórios para que  
    // seja diferente a cada execução  
    srand(time(NULL));  
  
    // gera um número aleatório entre 0 e RAND_MAX  
    valor = rand();  
  
    printf("Valor gerado: %d", valor);  
    getchar();  
}
```

No caso de querer gerar um número em um intervalo definido, basta seguir a seguinte regra:

```
rand() % (ValorMaximo-ValorMinimo) + ValorMinimo
```

Então, o exemplo abaixo gera um número entre 5 e 15:

```
rand() % 10 + 5
```

Exercícios com Funções de Biblioteca

1. Fazer um programa que leia três valores inteiros e encontre o maior deles a partir da fórmula matemática a seguir, mostrando-o com a mensagem "xxx é o maior". Dica: utilizar a função de biblioteca adequada para descobrir o valor absoluto.

$$\text{Maior entre } a \text{ e } b = \frac{a + b + |a - b|}{2}$$

2. Fazer um programa que leia três valores reais e calcule a média aritmética, harmônica e geométrica desses valores, mostrando os resultados. Dica: utilizar a função de biblioteca adequada para calcular a média geométrica, lembrando que uma radiciação pode ser transformada em potenciação.

$$\text{Média Aritmética} = \frac{a + b + c}{3}$$

$$\text{Média Harmônica} = \frac{3}{\frac{1}{a} + \frac{1}{b} + \frac{1}{c}}$$

$$\text{Média Geométrica} = \sqrt[3]{a \times b \times c}$$

3. Fazer um programa que gere um valor inteiro aleatório entre 1 e 10, utilizando a função de biblioteca adequada. Após, leia um valor inteiro e compare-o com o número gerado através do operador condicional, dando a mensagem adequada caso os valores sejam iguais ou não.

Precedência Entre os Operadores do C

A tabela a seguir mostra a ordem de precedência de todos os operadores vistos até agora. Os operadores de maior precedência são os **parênteses** e as **funções**. Os operadores de menor precedência são os operadores de **atribuição**.

Categoria	Operadores	Prioridade
parênteses	()	interno → externo
função	nome ()	E → D
incremental, lógico	++ -- !	E ← D
aritmético	* / %	E → D
aritmético	+ -	E → D
relacional	< > <= >=	E → D
relacional	== !=	E → D
lógico	&&	E → D

lógico	<code> </code>	$E \rightarrow D$
condicional	<code>?:</code>	$E \leftarrow D$
atribuição	<code>= += -= *= /= %=</code>	$E \leftarrow D$

Estruturas de Controle

Estruturas de controle permitem controlar a seqüência das ações lógicas de um programa. Basicamente, existem dois tipos de estruturas de controle: estruturas de **repetição** e estruturas de **decisão**. A estrutura de repetição permite que um bloco de instruções seja executado repetidamente uma quantidade controlada de vezes. A estrutura de decisão permite executar um entre dois ou mais blocos de instruções.

Condição de Controle

Em todas as estruturas, existe pelo menos uma expressão que faz o controle de **qual** bloco de instruções será executado ou **quantas vezes** ele será executado: é o que chamamos de **condição de controle**. Uma condição de controle é uma expressão lógica ou aritmética cujo resultado pode ser considerado verdadeiro ou falso. Como já foi visto, na linguagem C, uma **condição de controle** será considerada **falsa** se seu valor for **igual a zero**, e **verdadeira** se seu valor for **diferente de zero**.

Estrutura de Decisão `if...else`

A estrutura `if...else` é a mais simples estrutura de controle do C. Esta estrutura permite executar um entre vários blocos de instruções. O controle de qual bloco será executado será dado por uma **condição** (expressão lógica ou numérica).

A estrutura de decisão de um bloco apenas permite que se execute (ou não) um bloco de instruções conforme o valor de uma condição seja verdadeiro ou falso. **Sintaxe:**

```
if(condição)
{
    bloco de instruções
}
```

onde **condição** é uma expressão lógica ou numérica e **bloco de instruções** é um conjunto de instruções. Se a condição for **verdadeira**, o **bloco** é executado. Caso contrário, o **bloco** não é executado. Caso o bloco possua uma única instrução, não é necessária a utilização das chaves.

Exemplo: No trecho a seguir, se o valor lido for maior que 10, então o seu valor é redefinido como 10. Observe que o bloco constitui-se de um única instrução.

```
printf("Digite um valor menor ou igual a 10: ");
scanf("%d",&iter);
```

```
fflush(stdin);
if(iter > 10)
    iter = 10;
```

Percebe-se que, como o **bloco de instruções** possui uma única instrução, não é necessária a utilização das chaves. O exemplo a seguir mostra uma situação onde a utilização das chaves é necessária.

```
printf("Digite um valor menor ou igual a 10: ");
scanf("%d",&iter);
fflush(stdin);
if(iter > 10)
{
    printf("O valor digitado foi maior que 10");
    iter = 10;
}
```

Também é possível escrever uma estrutura que execute um entre dois blocos de instruções.
Sintaxe:

```
if(condição)
{
    bloco 1;
}
else
{
    bloco 2;
}
```

Se a **condição** for **verdadeira** o **bloco 1** é executado. Caso contrário, o **bloco 2** é executado. **Exemplo:** No trecho a seguir, se o valor de **raiz*raiz** for maior que **num** o valor de **raiz** será atribuído a **max**, caso contrário, será atribuído a **min**.

```
if(raiz*raiz > num)
    max = raiz;
else
    min = raiz;
```

Novamente, percebe-se que, como os dois blocos possuem uma única instrução, não é necessária a utilização das chaves. O exemplo a seguir mostra uma situação onde a utilização das chaves é necessária.

```
if(raiz*raiz > num)
{
    printf("o valor sera atribuido a max");
    max = raiz;
}
else
{
    printf("o valor sera atribuido a min");
    min = raiz;
}
```

Também é possível escrever uma estrutura que execute um entre múltiplos blocos de instruções. **Sintaxe:**

```
if(condição 1)
{
    bloco 1;
}
else if(condição 2)
{
    bloco 2;
}
else if(condição N)
{
    bloco N;
}
else
{
    bloco P
}
```

Se a **condição 1** for **verdadeira**, o **bloco 1** é executado. Caso contrário, a **condição 2** é avaliada. Se a **condição 2** for **verdadeira**, o **bloco 2** é executado. Caso contrário, a **condição 3** é avaliada, e assim sucessivamente. Se nenhuma condição for **verdadeira**, o **bloco P** é executado. Observa-se que apenas um dos blocos é executado.

Exemplo: No trecho a seguir, uma determinada ação é executada se o valor de **num** for positivo, negativo ou nulo.

```
if(num > 0)
    a = b;
else if(num < 0)
    a = b + 1;
else
    a = b - 1;
```

Exercícios com a Estrutura de Controle if...else

1. Fazer um programa que leia três números inteiros e imprima o número de maior valor absoluto, com a mensagem "O maior valor absoluto é xxx".
2. Fazer um programa que leia um valor inteiro de 0 a 100 escreva o seu valor por extenso. Por exemplo:

```
Digite valor: 79
Extenso: setenta e nove
```

3. Fazer um programa que leia três valores quaisquer e verifica se eles formam ou não um triângulo. Supor que os valores lidos são inteiros e positivos. Caso os valores formarem um triângulo, calcular e mostrar a área deste triângulo. Se não formarem, mostrar os valores lidos e a mensagem "não formam triângulo".

$$\text{Area} = \sqrt{s(s-a)(s-b)(s-c)}$$

onde s é o semi-perímetro $s = \frac{a+b+c}{2}$

4. Fazer um programa que leia a hora de início de um jogo e a hora de término do jogo, ambas subdivididas em dois valores distintos, a saber: horas e minutos. Calcular e mostrar a duração do jogo, também em horas e minutos, considerando que o tempo máximo de duração de um jogo pode iniciar em um dia e terminar no dia seguinte.

Estrutura de Controle `switch...case`

A estrutura `switch...case` é uma estrutura de decisão que permite a execução de um bloco de instruções a partir de pontos diferentes, conforme o resultado de uma expressão de controle (que pode ser inteira ou caracter). O resultado desta expressão é comparado ao valor de cada um dos rótulos, e as instruções são executadas a partir desde rótulo. **Sintaxe:**

```
switch(expressão)
{
    case rótulo_1: bloco 1
    case rótulo_2: bloco 2
    ...
    case rótulo_n: bloco n
    [default: bloco d]
}
```

onde `expressão` é uma expressão inteira ou caracter, `rótulo_1`, `rótulo_2`, `rótulo_n` e `rótulo_d` são constantes inteiras ou caracter e `bloco 1`, `bloco 2`, `bloco n` e `bloco d` são blocos de instruções.

O valor de expressão é avaliado e o fluxo lógico será desviado para o bloco cujo `rótulo` é igual ao resultado da expressão e todas as instruções **abaixo** deste rótulo serão executadas. Caso o resultado da expressão for diferente de todos os valores dos rótulos, então `bloco d` é executado. Os rótulos devem ser expressões constantes inteiras ou caracter **diferentes** entre si. O rótulo `default` é opcional.

Esta estrutura é particularmente útil quando se tem um bloco de instruções que se deve executar em ordem, porém se pode começar em pontos diferentes.

Exemplo: O trecho a seguir ilustra o uso da instrução `switch` em um menu de seleção. Neste exemplo, o programa iniciará o processo de usinagem de uma peça em um ponto qualquer, dependendo do valor lido.

```
int selecao;
printf("Digite estagio de usinagem:");
scanf("%d",&selecao);
fflush(stdin);
switch(selecao)
{
    case 1: printf("Será realizado um desbaste grosso");
```

```

        break;
    case 2: printf("Será realizado um desbaste fino");
            break;
    case 3: printf("Será realizado o acabamento");
            break;
    case 4: printf("Será realizado o polimento");
            break;
    default: printf("Estágio inexistente");
}

```

Percebe-se a existência da instrução `break` ao final de cada bloco de instruções. Isso é necessário, pois o `switch...case` não utiliza chaves para delimitação do início/fim de cada bloco.

Exercícios com a Estrutura de Controle `switch...case`

1 - Fazer um programa que faça a leitura de três variáveis `float`. Após, mostrar o menu:

```

1- Mostrar os valores em ordem crescente
2- Mostrar os valores em ordem decrescente

```

Ler uma variável `opcao` (`int`) que será a opção selecionada pelo usuário. Conforme a opção, realizar a tarefa. Obrigatório: Utilizar o `if` para a ordenação e o `switch` para testar a opção.

2 - Fazer um programa que emule uma calculadora básica. Ler um primeiro valor (`float`), um operador (`char`) e um segundo valor (`float`). Conforme o operador, que pode ser `+`, `-`, `*`, `/`, executar a operação e mostrar o resultado. Obrigatório: Utilizar o `switch` para testar o operador.

3 - Escrever um programa que leia uma variável do tipo `char` e, conforme o caracter armazenado, (`'('`, `')`, `'['`, `']'`, `'{'` e `'}'`), imprime `ABRE_PARENTESES`, `FECHA_PARENTESES`, `ABRE_COLCHETES`, `FECHA_COLCHETES`, `ABRE_CHAVES`, `FECHA_CHAVES`, respectivamente. Obrigatório: Utilizar o `switch`.

Estrutura de Repetição `do...while`

Esta é uma estrutura básica de repetição condicional. Permite a execução de um bloco de instruções repetidamente. **Sintaxe:**

```

do{
    bloco de instruções
}while(condição);

```

onde `condição` é uma expressão lógica e `bloco de instruções` é um conjunto de instruções.

Esta estrutura faz com que o bloco de instruções seja executado pelo menos uma vez. Após a execução do bloco, a condição é avaliada. Se a condição for **verdadeira** o bloco é executado outra vez, caso contrário a repetição é terminada.

Exemplo 1: No trecho a seguir, uma variável inicializada com zero é incrementada em um e mostrada enquanto for menor que cinco.

```
a = 0;
do{
    a = a + 1;
    printf("%d",a);
}while(a < 5);
```

Exemplo 2: No trecho a seguir, a leitura de um número é feita dentro de um laço de repetição condicional. A leitura é repetida caso o número lido seja negativo.

```
do{
    printf("Digite um número positivo: ");
    scanf("%f",&num);
    fflush(stdin);
    if (num <= 0)
        printf("Valor inválido!")
}while(num <= 0);
```

Estrutura de Repetição `while`

A estrutura de repetição condicional `while` é semelhante à estrutura `do...while`.
Sintaxe:

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

onde `condição` é uma expressão lógica ou numérica e `bloco de instruções` é um conjunto de instruções.

Esta estrutura faz com que a condição seja avaliada em primeiro lugar. Se a condição for **verdadeira** o bloco é executado uma vez e a condição é avaliada novamente. Caso a condição seja **falsa** a repetição é terminada sem a execução do bloco. Observa-se que, nesta estrutura, ao contrário da estrutura `do...while`, o bloco de instruções pode não ser executado nenhuma vez, basta que a condição seja inicialmente falsa.

Exemplo: No trecho a seguir, uma variável inicializada com zero é incrementada em um e mostrada enquanto for menor que cinco.

```
a = 0;
while(a < 5)
{
```

```

    a = a + 1;
    printf("%d",a);
}

```

Exercícios com as Estruturas de Repetição `do...while` e `while`

1. Fazer um programa que faça a leitura de um par de valores `a` e `b`, inteiros e positivos (deve ter teste na leitura – utilizar o `do-while`), e escreve os números pares existentes entre o menor e o maior destes números lidos, incluindo-os se forem pares.
2. A partir da sucessão a seguir, fazer um programa que faça a leitura do termo de ordem desejado, mostrando-o e logo após solicitando se deseja mais algum termo. Se a resposta for negativa, encerrar o programa.

ORDEM:	1	2	3	4	5	6	7	8	...
SUCESSÃO:	-9	-7	-1	1	7	9	15	17	...

Estrutura de Repetição `for`

A estrutura `for` é muito semelhante às estruturas de repetição vistas anteriormente, entretanto costuma ser utilizada quando se quer um número determinado de ciclos. A contagem dos ciclos é feita por uma variável chamada de **contador**. **Sintaxe:**

```

for(inicialização; condição; incremento)
{
    bloco de instruções
}

```

onde `inicialização` é uma expressão de inicialização do contador, `condição` é uma expressão lógica de controle de repetição, `incremento` é uma expressão de incremento do contador e `bloco de instruções` é um conjunto de instruções a ser executado.

Esta estrutura executa um número determinado de repetições usando um contador de iterações. O contador é inicializado na expressão de `inicialização` **antes** da primeira iteração. Por exemplo: `i = 0;` ou `cont = 20;`. Então, o bloco é executado e **depois** de cada iteração, o contador é incrementado de acordo com a expressão de `incremento`. Por exemplo: `i++` ou `cont -= 2`. Então a expressão de condição é avaliada: se a condição for verdadeira, o `bloco de instruções` é executado novamente e o ciclo recomeça; se a condição for falsa termina-se o laço. Esta condição é, em geral, uma expressão lógica que determina o último valor do contador. Por exemplo: `i <= 100` ou `cont > 0`.

Exemplo: No trecho a seguir, uma variável inicializada com `1` é automaticamente incrementada em `1` e mostrada enquanto for menor ou igual a cinco.

```

for(a = 1 ; a <= 5 ; a++)
    printf("%d",a);

```

Exemplo comparativo: As seguintes instruções são plenamente equivalentes:

```

i = 0;
do{
    bloco de instruções
    i++;
}while(i <= 100);

for(i = 0; i <= 100; i++)
{
    bloco de instruções
}

```

Exercício com a Estrutura de Repetição for

1. Fazer um programa que gera e mostra os 4 primeiros números perfeitos. Um número perfeito é aquele que é igual a soma dos seus divisores (Ex.: $6=1+2+3$; $28=1+2+4+7+14$). Utilizar o `for`.

Instruções de Interrupção e Desvio: `break` - `continue` - `exit()`

As instruções vistas anteriormente podem sofrer **desvios** e **interrupções** em sua seqüência lógica normal através do uso certas instruções.

A Instrução `break`

Esta instrução serve para terminar a execução das instruções de um laço de repetição (`for`, `do...while`, `while`) ou para terminar um conjunto `switch...case`. Quando em um laço de repetição, esta instrução força a interrupção do laço independentemente da condição de controle.

Exemplo: No trecho a seguir, um laço de repetição lê valores para o cálculo de uma média. O laço possui uma condição de controle sempre verdadeira, o que constitui um laço infinito que deve ter um término. Isso se dá pela instrução `break`, que é executada quando um valor negativo é lido.

```

do{
    printf("Digite um valor (-1 para sair):");
    scanf("%d",&val);
    fflush(stdin);
    if(val == -1)
        break;          /* saída do laço */
    num++;
    soma += val;
}while(1);              /* sempre verdadeiro - laço infinito */
printf("A média é: %f",soma/num);

```

A Instrução `continue`

Esta instrução opera de modo semelhante à instrução `break` dentro de um laço de repetição. Quando executada, ela pula as instruções de um laço de repetição sem sair do laço. Isto é, a instrução força a avaliação da condição de controle do laço.

Exemplo: No trecho a seguir, tem-se um laço de repetição que lê valores para o cálculo de uma média. Se o valor for menor que zero, o programa salta diretamente para a condição de controle, sem executar o resto das instruções. Se o valor for `-1`, o laço é encerrado.

```
do{
    printf("Digite valor: ");
    scanf("%d",&val);
    fflush(stdin);
    if(val == -1)
        break;          /* saída do laço */
    if(val < 0)           /* se val é negativo... */
    {
        printf("O valor deve ser positivo");
        continue;       /* ...salta para o fim do laço... */
    }
    num++;               /* ...e essas instruções... */
    soma += val;         /* ...não são executadas! */
}while(1);
printf("A média é: %f",soma/num);
```

A Função `exit()`

A função `exit()`, da biblioteca `stdlib.h`, termina a execução de um programa.

Exemplo: No trecho a seguir, tem-se um laço de repetição que lê valores para o cálculo de uma média. Se o valor for igual a `-1`, a média é mostrada e logo após o laço é encerrado.

```
do{
    printf("Digite valor: ");
    scanf("%d",&val);
    fflush(stdin);
    if(val == -1)           /* se val é negativo... */
    {
        printf("A média é: %f",soma/num); /* ...imprime resultado... */
        exit(0);              /* ...e termina o programa */
    }
    num++;
    soma += val;
}while(1);
```

Exercícios Diversos

1. Fazer um programa que faça a leitura de três valores inteiros `a`, `b` e `c` e calcule e mostre a soma dos inteiros de `a`, inclusive, até `b`, com uma variação igual a `c`. Exemplo: Se `a=5`, `b=19` e `c=3`, então `soma=5+8+11+14+17=55`. Após a execução, fazer a leitura de `a`, `b` e `c` novamente e, se for digitado `-1` em `a`, encerrar a aplicação.
2. O cardápio de uma casa de lanches é dado pela tabela a seguir:

Código	Especificação	Preço Unitário
100	Cachorro-quente	3,50

101	Bauru Simples	4,00
102	Bauru com ovo	4,50
103	Hambúrguer	5,50
104	Cheese burger	6,00
105	Refrigerante	1,50
106	Cerveja	2,50

Fazer um programa que mostre o menu e leia o nome do cliente, o código (de 100 a 106) e a quantidade de cada item adquirido. A cada item digitado, fazer a pergunta: *Deseja inserir mais algum item para esse cliente?*. Se a resposta for negativa, calcular e mostrar o valor da conta a pagar por aquele cliente, assim: *A conta do cliente FULANO deu xxx,xx*. A seguir, perguntar: *Deseja calcular a conta de mais algum cliente?* Encerrar o programa apenas se a resposta for negativa. Dica: Utilizar as instruções de interrupção e desvio.

- Fazer um programa que peça para o usuário adivinhar um número escolhido aleatoriamente entre 1 e 20. Se o usuário digitar um número errado, o programa responde o novo intervalo do número procurado. Se o usuário acertou o número procurado, o programa mostra quantos palpites foram dados e faz a pergunta: *Deseja jogar novamente?* Encerrar o programa apenas se a resposta for negativa. Por exemplo:

```
O número procurado está entre 1 e 20:
Palpite: 13
O número procurado está entre 1 e 12:
Palpite: 5
O número procurado está entre 6 e 12:
Palpite: 9

Parabéns! Você acertou o número em 3 tentativas.

Deseja jogar novamente?
```

- Fazer um programa que emule uma calculadora elementar, com as operações de soma ('+'), subtração ('-'), divisão ('/') e multiplicação ('*'), além do 'f' para finalizar. Testar o operador para evitar entradas inválidas. Exemplo de funcionamento:

```
Digite valor: 10
Digite operador(+ - * / f): +
Digite valor: 5
Resultado: 15
Digite operador(+ - * / f): -
Digite valor: 2
Resultado: 13
Digite operador(+ - * / f): x
Valor inválido
Digite operador(+ - * / f): *
Digite valor: 2
Resultado: 26
Digite operador(+ - * / f): f
```

Vetores

Em muitas aplicações, é necessário trabalhar com conjuntos de dados que são **semelhantes em tipo**, por exemplo: o conjunto das alturas dos alunos de uma turma, ou um conjunto de seus nomes. Nestes casos, seria conveniente poder colocar estas informações sob um mesmo conjunto, e poder referenciar cada dado individual deste conjunto por um índice. Em programação, este tipo de estrutura de dados é chamada de **vetor** (ou *array*, em inglês) ou, de maneira mais formal, **estruturas de dados homogêneas**. **Exemplo:** A maneira mais simples de entender um vetor é através da visualização de um **lista** de elementos com um nome coletivo e um índice de referência aos valores da lista.

```
n   nota
0   8.4
1   6.9
2   4.5
3   4.6
```

Nesta lista, `n` representa um número de referência e `nota` é o nome do conjunto. Assim, podemos dizer que a 2ª nota é `6.9`, ou representar `nota[1] = 6.9`

Esta não é a única maneira de estruturar conjuntos de dados. Também é possível organizar dados sob forma de tabelas. Neste caso, cada dado é referenciado por dois índices e diz-se que se trata de um **vetor bidimensional** (ou **matriz**).

Declaração de vetores

Em C, um vetor é um conjunto de variáveis de um **mesmo tipo** que possuem um nome identificador e um índice de referência. **Sintaxe:**

```
tipo nome[tam];
```

onde `tipo` é o **tipo** dos elementos do vetor: `int`, `float`, `double`, ...; `nome` é o **nome** identificador do vetor. As regras de nomenclatura de vetores são as mesmas usadas em variáveis, e; `tam` é o tamanho do vetor, isto é, o número de elementos que o vetor pode armazenar. **Exemplos:**

```
int idade[100]; // declara um vetor chamado 'idade' do tipo
                // 'int' que recebe 100 elementos.
float nota[25]; // declara um vetor chamado 'nota' do tipo
                // 'float' que pode armazenar 25 números.
char nome[80];  // declara um vetor chamado 'nome' do tipo
                // 'char' que pode armazenar 80 caracteres.
```

Referência a elementos de vetor

Cada elemento do vetor é referenciado pelo **nome** do vetor seguido de um **índice** inteiro. O **primeiro** elemento do vetor tem índice `0` e o **último** tem índice `tam-1`. O índice de um vetor deve ser **inteiro**.

Exemplo: Algumas referências a vetores:

```
int i = 7;
float valor[10];           /* declaração de vetor */
valor[1] = 6.645;
valor[i] = 7.645;
```

Inicialização de vetores

Assim como é possível inicializar variáveis (por exemplo: `int j = 3;`), pode-se inicializar vetores. **Sintaxe:**

```
tipo nome[tam] = {lista de valores};
```

onde `lista de valores` é uma lista, separada por vírgulas, dos valores de cada elemento do vetor.

Exemplo:

```
int dia[7] = {12,30,14,7,13,15,6};
float nota[5] = {8.4,6.9,4.5,4.6,7.2};
char vogal[5] = {'a', 'e', 'i', 'o', 'u'};
```

Opcionalmente, é possível inicializar os elementos do vetor enumerando-os um a um.

Exemplo: A duas inicializações a seguir são possíveis:

```
int cor_menu[4] = {1,2,3,4};
```

ou

```
int cor_menu[4];
cor_menu[0] = 1;
cor_menu[1] = 2;
cor_menu[2] = 3;
cor_menu[3] = 4;
```

Limites

Na linguagem C deve-se ter cuidado com os limites de um vetor. Embora na sua declaração se defina o tamanho de um vetor, o C não faz nenhum teste de verificação de acesso a um elemento no vetor. Por exemplo: se for declarado um vetor como `int valor[5]`, teoricamente só tem sentido usar os elementos `valor[0]`, ..., `valor[4]`. Porém, o C não acusa **erro** se for usado `valor[12]` em algum lugar do programa. Estes testes de limite **devem** ser feitos **logicamente** dentro do programa.

Acessar um segmento fora do espaço destinado a um vetor pode **destruir informações** reservadas de outras variáveis. Estes erros são difíceis de detectar, pois o compilador não gera nenhuma mensagem de erro. A solução mais adequada é sempre avaliar os limites de um vetor antes de manipulá-lo.

Vetores Multidimensionais

Vetores podem ter mais de uma dimensão, isto é, mais de um índice de referência. Pode-se ter vetores de duas, três, ou mais dimensões. **Exemplo:** Um vetor bidimensional pode ser visualizado através de uma **tabela**.

nota	0	1	2
0	8.4	7.4	5.7
1	6.9	2.7	4.9
2	4.5	6.4	8.6
3	4.6	8.9	6.3
4	7.2	3.6	7.7

Na tabela estão representadas as notas de 5 alunos em 3 provas diferentes. O nome `nota` é o nome do conjunto, assim pode-se dizer que a nota do 3º aluno na 2ª prova é `6.4` ou representar `nota[2][1] = 6.4`.

Declaração e inicialização

A declaração e inicialização de vetores de mais de uma dimensão é feita de modo semelhante aos vetores unidimensionais. **Sintaxe:**

```
tipo nome[tam_1][tam_2]...[tam_N]={ {lista}, {lista}, ... {lista} };
```

onde `tipo` é o tipo dos elementos do vetor, `nome` é o nome do vetor, `[tam_1][tam_2]...[tam_N]` é o tamanho de cada dimensão do vetor, e `{ {lista}, {lista}, ... {lista} }` são as listas de valores da inicialização.

Exemplo:

```
float nota[5][3] = { {8.4, 7.4, 5.7},  
                    {6.9, 2.7, 4.9},  
                    {4.5, 6.4, 8.6},  
                    {4.6, 8.9, 6.3},  
                    {7.2, 3.6, 7.7} };  
int tabela[2][3][2] = { { {10, 15}, {20, 25}, {30, 35} },  
                       { {40, 45}, {50, 55}, {60, 65} } };
```

Observações: Algumas observações a respeito de vetores multidimensionais podem ser feitas:

- Os índices dos vetores multidimensionais, também começam em 0. Por exemplo: `vet[0][0]`, é o primeiro elemento do vetor.

- Embora uma tabela não seja a única maneira de visualizar um vetor bidimensional, podemos entender o **primeiro** índice do vetor como o índice de **linhas** da tabela e o **segundo** índice do vetor como índice das **colunas**.

Exercícios sobre Vetores

1. Criar um programa que faça a leitura de valores inteiros e positivos para um vetor de 6 posições. Em seguida, declarar um outro vetor qualquer de 6 posições do tipo `float`. Declarar também uma variável `float` qualquer, fazendo a leitura dela. Após, trocar o primeiro valor do vetor de inteiros com o último, o segundo com o penúltimo e o terceiro com o antepenúltimo. Por fim, preencher o vetor `float` com os elementos do vetor de inteiros na ordem inversa multiplicados pelo valor da variável `float`. Mostrar, ao final, os elementos de cada um dos vetores.
2. Criar um programa que faça a leitura de valores inteiros e positivos para um vetor de 7 posições. Em seguida, ordene o vetor, mostrando-o. Após, criar uma matriz do tipo `float 2x7` e preenchê-la, na primeira linha, com o quadrado de cada um dos sete elementos do vetor e, na segunda linha, com o cubo de cada um dos sete elementos do vetor. Mostrar a matriz ao final.
3. Criar um programa que faça a leitura de uma matriz `10x10` com valores inteiros. Troque, a seguir, a linha 2 com a linha 8, a coluna 4 com a coluna 10, a diagonal principal com a secundária e a linha 5 com a coluna 10. Mostre a matriz modificada.

Produto Matricial

O produto matricial é obtido multiplicando-se uma matriz $A(m,n)$ por uma matriz $B(n,x)$, sendo que o número de colunas de $A(n)$ deve ser igual ao número de linhas de $B(x)$. Esta multiplicação originará uma matriz $M(m,x)$.

Para calcular o produto de uma matriz $A(m,n)$ por uma matriz $B(n,x)$ devemos seguir os seguintes passos:

1. Verificar se o número de linhas da matriz B é igual ao número de colunas da matriz A. Se não for, o cálculo não poderá ser realizado.
2. Gerar uma matriz temporária $T(m,x)$ onde ficarão armazenados os produtos da multiplicação dos elementos da matriz A pelos elementos da matriz B.
3. Calcular os elementos da matriz T, onde:
 - a. A matriz T será de ordem (m,x) , ou seja, terá o número de linhas da matriz A e o número de colunas da matriz B.
 - b. O elemento $T(i,j)$ é igual ao somatório da multiplicação dos elementos da linha i da matriz A pela coluna j da matriz B;

Exemplo:

$A = \begin{pmatrix} 13 & 18 & 20 \\ 2 & 3 & 4 \end{pmatrix}$	$T[0,0] = 13 \times 12 + 18 \times 24 + 20 \times 12 = 828$ (linha 0 de A com coluna 0 de B)	$T[0,1] = 13 \times 6 + 18 \times 12 + 20 \times 9 = 474$ (linha 0 de A com coluna 1 de B)
	$T[1,0] = 2 \times 12 + 3 \times 24 + 4 \times 12 = 144$ (linha 1 de A com coluna 0 de B)	$T[1,1] = 2 \times 6 + 3 \times 12 + 4 \times 9 = 84$ (linha 1 de A com coluna 1 de B)
	$B = \begin{pmatrix} 12 & 6 \\ 24 & 12 \\ 12 & 9 \end{pmatrix}$	

Exercícios sobre Produto Matricial

1. Criar um programa que faça a leitura de duas matrizes e depois calcule o produto matricial, armazenando em outra matriz, mostrando-a. Para conferir a correção, utilize os tamanhos e valores do exemplo anterior.

Funções

Funções, também chamadas de **rotinas**, ou **sub-programas**, são a essência da programação estruturada. Funções são segmentos de programa que executam uma determinada tarefa específica. Já foram vistas funções nas seções anteriores: funções existentes nas bibliotecas-padrão do C (como `sqrt()`, `toupper()`, `getchar()` ou `putchar()`).

O programador pode escrever suas próprias rotinas. São as chamadas **funções de usuário** ou rotinas de usuário. Deste modo pode-se segmentar um programa grande em vários programas menores. Esta segmentação é chamada de **modularização** e permite que cada segmento seja escrito, testado e revisado individualmente, sem alterar o funcionamento do programa como um todo.

Estrutura das funções de usuário

A estrutura de uma função de usuário é muito semelhante a estrutura dos programas que escrevemos até agora. Uma função de usuário constitui-se de um **bloco de instruções** que definem os procedimentos efetuados pela função, um **nome** pelo qual a chamamos, uma **lista de argumentos** passados à função e o tipo de dado de **retorno**. Chamamos este conjunto de elementos de **definição da função**. Sintaxe:

```
tipo_de_retorno nome_da_função(tipo_1 arg_1, tipo_2 arg_2, ...)
```

```

    [bloco de instruções da função]
}

```

Exemplo: o código mostrado a seguir é uma função definida pelo usuário para calcular a média aritmética de dois números reais:

```

float media(float a, float b)
{
    float med;
    med = (a + b) / 2.0;
    return(med);
}

```

No exemplo acima, foi definida uma função chamada `media` que recebe dois argumentos tipo `float`: `a` e `b`. A média destes dois valores é calculada e armazenada na variável `med`, declarada internamente. Quando uma variável é declarada internamente, ela é chamada de **variável local** e seu escopo, ou sua visibilidade se restringe àquela função. A função retorna, para o local que a chamou, um valor também do tipo `float`: o valor da variável `med`. Este retorno de valor é feito pela função `return()`. Depois de definida, a função pode ser usada dentro de uma outra função, ou até mesmo do `main()`. Diz-se que está sendo feita uma **chamada** à função.

Importante: No C++ da Microsoft é obrigatória a definição da função antes de sua chamada, ou seja, não é possível invocar uma função que está descrita após à sua invocação. Outros compiladores permitem essa situação.

Exemplo: No exemplo a seguir, é chamada a função `media()` dentro da função principal:

```

void main()
{
    float num_1, num_2, medarit;
    puts("Digite dois números: ");
    scanf("%f %f", &num_1, &num_2);
    fflush(stdin);
    medarit = media(num_1, num_2); /* chamada a função */
    printf("\nA media destes números é' %f", medarit);
}

```

Percebe-se que a função `main()` é agora declarada com retorno `void`. O `void` indica que a função não tem retorno algum. Importante: Não é possível retornar mais do que um valor de uma função. Quanto aos parâmetros, não existe limite quanto ao número e, quando não houver nenhum parâmetro, simplesmente colocam-se apenas os parênteses. Exemplo: `int calculo()`. Normalmente, as funções são escritas após a função `main()`.

Em alguns compiladores há a exigência de que as funções sejam declaradas antes de serem usadas. As declarações vão depois dos `#include`. A declaração da função `float media(float a, float b)` ficaria assim:

```

float media(float, float);

```

Já outros compiladores, como o C++ da Microsoft não exige isso, mas não há problema se for feita a declaração.

Exercícios sobre Funções

1. Criar quatro funções: `soma`, `subtracao`, `multiplicacao` e `divisao` que devem receber como parâmetro dois valores `float`, fazer a operação correspondente e retornar o valor calculado (`float`). Após, na função `main()`, ler um valor `float v1`, um operador `char op`, que pode ser `+`, `-`, `*`, `/`, e um segundo valor `float v2`. As variáveis devem ser locais do `main()`. Através do `switch`, testar o operador e, conforme o caso, chamar uma das quatro funções criadas: `soma`, `subtracao`, `divisao`, `multiplicacao`, mostrando o resultado na tela.
2. Alterar o programa do exercício anterior para que funcione como uma calculadora elementar, incluindo o operador `f` para sair. Fazer testes na leitura do operador. Ex.:

```
Digite valor: 10
Digite operador(+ - * / f): +
Digite valor: 5
Resultado: 15
Digite operador(+ - * / f): -
Digite valor: 2
Resultado: 13
Digite operador(+ - * / f): x
Valor inválido
Digite operador(+ - * / f): *
Digite valor: 2
Resultado: 26
Digite operador(+ - * / f): f
```

3. Criar uma função `verif_cpf` que recebe um parâmetro `char*` e testa o dígito verificador. A função deve retornar um valor `int`, que será `1` se o dígito for válido e `0` caso contrário. No `main()`, fazer a leitura de um valor `char` de 11 posições. Se o valor tiver as 11 posições preenchidas (utilizar o `strlen()` - precisa incluir a biblioteca `string.h` - para testar o tamanho do string), enviar como parâmetro para o `verif_cpf` e se o retorno for `1`, mostrar "CPF Válido", caso contrário, mostrar "CPF Inválido". O algoritmo de cálculo do dígito verificador do CPF está a seguir. Dica: Como o que vai ser recebido como parâmetro é um vetor de `char`, deve-se trabalhar com cada índice, convertendo para `int`. Ex. se o valor passado for "12345678909" o primeiro valor será o `vetor[0]-48`, ou seja '1'- 48, que vai resultar em 1 inteiro.

Algoritmo para Validação do CPF:

- **Descrição:** O CPF (Cadastro de Pessoas Físicas) serve para identificar cada indivíduo no país. O número do CPF é composto de 11 dígitos, sendo os dois últimos os dígitos de verificação. A fórmula para verificar a validade do número do CPF é simples e é explicada a seguir.

Tomando como exemplo o número 123.456.789-09.

• 1º Dígito Verificador

Primeiro é calculada a soma da multiplicação dos 9 primeiros dígitos por 10, 9, 8, ..., 3, 2, respectivamente. Ou seja:

```
soma = (1*10) + (2*9) + ... + (8*3) + (9*2);
```

Em seguida, se obtém o resto da divisão de Soma por 11, através do operador específico, assim:

```
resultado = soma % 11;
```

Se `resultado` for igual à 1 ou à 0, então o 1º dígito verificador é 0. Caso contrário, o 1º dígito verificador é o resultado da subtração de 11 por `resultado`.

• 2º Dígito Verificador

Primeiro é calculada a soma da multiplicação dos 9 primeiros dígitos por 11, 10, 9, ..., 4, 3, respectivamente, e em seguida é somado com (`digito1*2`), sendo que `digito1` é o valor encontrado para o 1º dígito verificador. Ou seja:

```
Soma = (1*11) + (2*10) + ... + (8*4) + (9*3) + (Digito1*2);
```

O resto é semelhante ao que foi feito anteriormente:

```
resultado = soma % 11;
```

Agora analisamos Resultado:

Se `resultado` for igual à 1 ou à 0, então o 2º dígito verificador é 0. Caso contrário, o 2º dígito verificador é o resultado da subtração de 11 por `resultado`.

No exemplo apresentado, (123.456.789-09) possui dígito verificador válido.

4. Fazer duas funções:

- Função para consistência de uma data. Deverá receber três parâmetros inteiros e retornar `false` se a data passada como argumento estiver errada e `true` se ok. Dica: utilizar um vetor contendo os últimos dias de cada mês.

Protótipo: `bool consiste_data(int dia, int mes, int ano)`

Exemplo: `consiste_data(30,2,90)` deverá retornar 0 pois é uma data inválida.

- Função que receba, por parâmetro, uma data particionada em dia, mês e ano. A função deverá retornar o número do dia no ano. Dica: utilizar um vetor contendo os últimos dias de cada mês.

Protótipo: `int data_nrodia(int dia, int mes, int ano)`

Exemplo: `data_nrodia(31,12,92)` deverá retornar 366.

Após, no `main()` do programa, mostrar o menu:

1. Entrada de dados
2. Verificação da validade
3. Retorno do número do dia no ano
4. Encerrar

A opção 1 deve solicitar a digitação da data (três variáveis inteiras). Se a opção for 2, passar os valores digitados por parâmetro para o `consiste_data` e, conforme a data for válida ou não, mostrar a mensagem adequada. Se a opção for 3, passar os valores digitados por parâmetro para o `data_nrodia` e mostrar o retorno na tela. A opção 4 encerra a aplicação.

Manipulação de Strings

Em C, uma string é um conjunto de caracteres armazenados num vetor. As strings são representadas utilizando aspas enquanto os caracteres entre apóstrofes. Definitivamente, o forte do C não é o tratamento direto de strings, visto que strings não são tipos básicos da linguagem e a única forma de representar um conjunto de caracteres é recorrendo a um vetor.

Por exemplo não podemos fazer:

- uma atribuição `s1="OLA";`
- uma comparação `if (s1=="OLA") ...`
- uma concatenação `s1 = "OLA" + " MUNDO";`

Uma exceção é no momento da declaração, onde é possível realizar uma atribuição de valor:

```
char nome[]="OLA";
```

No entanto, C possui uma poderosa biblioteca de funções que permite realizar estas tarefas e muito mais, e se por acaso não encontrarmos nenhuma função que faça o que queremos podemos sempre fazer manipulação caracter a caracter e resolver o problema (algumas

linguagens tratam strings como um todo e não permitem manipular como um vetor de caracteres).

A declaração de strings obedece à sintaxe de declaração de vetores, sendo a primeira posição 0 e o final da string identificado pelo caracter '\0'. Isso é importante, pois como já foi visto, o C não faz tratamento de final de vetor, tendo, por isso, que delimitar o final de uma string. E isso é feito com o '\0'. Alguns compiladores aceitam também o termo `NULL` no lugar do '\0'.

Observe o exemplo a seguir. É possível fazer a atribuição utilizando uma função específica (`strcpy()`), ou simplesmente adicionando caracter a caracter nas posições adequadas, não esquecendo do '\0'.

```
#include <stdafx.h>
#include <stdio.h>
#include <string.h>

void main()
{
    char string1[10], string2[10];

    printf("\nAtribuição de valor a strings\n");
    printf("-----\n");
    strcpy(string1, "Ola");
    printf("\nAtribuição na primeira forma: %s", string1);
    string2[0] = 'O';
    string2[1] = 'l';
    string2[2] = 'a';
    string2[3] = '\0'; // ou string2[3] = NULL;
    printf("\nAtribuição na segunda forma: %s", string2);
    getchar();
}
```

Então, como visto, no C existem várias funções que realizam tratamento de vetores de caracter, ou strings. Serão abordadas agora algumas funções, destacando sua utilidade, com pequenos exemplos no final de cada uma.

Determinando o tamanho de uma string

Para determinar o tamanho de uma string use a função `strlen()`. Esta função faz parte da biblioteca `string.h`. Sua sintaxe é:

```
int strlen(char *str)
```

Exemplo:

```
/* Determinando o tamanho de uma string usando
 * a função strlen() */

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

```

#include <string.h>

void main()
{
    char string[20];

    printf("\nDeterminando o tamanho de uma string\n");
    printf("-----\n\n");
    printf("Digite a string: ");
    gets(string);
    printf("\nA string tem %d caracteres.\n\n",strlen(string));
    getchar();
}

```

Copiando strings

Para copiar uma string em outra use a função `strcpy()`. Esta função faz parte da biblioteca `string.h`. Sua sintaxe é:

```
char* strcpy(char* destino, char* origem)
```

Caso se deseje copiar apenas um número determinado de caracteres, partindo do início, a função a ser utilizada é a `strncpy_s()`. Sua sintaxe é:

```
char* strncpy_s(char* destino, char* origem, int numrerocaracteres)
```

Observação: pode-se utilizar o `strncpy()` no lugar do `strncpy_s()`, porém, é necessário colocar o `'\0'` no final da string copiada.

Exemplo:

```

/* Copiando uma string em outra usando as
 * funções strcpy() e strncpy_s() */

#include <stdafx.h>
#include <stdio.h>
#include <string.h>

void main()
{
    char string1[10], string2[10], string3[10];

    printf("\nCopiando strings\n");
    printf("-----\n\n");
    printf("Digite string1: ");
    gets(string1);
    printf("\nstring1 = %s\n",string1);
    strcpy(string2,string1);
    printf("string2 = %s\n",string2);
    strncpy_s(string3,string1,3);
    printf("string3 = %s\n",string3);
    getchar();
}

```

Unindo duas strings (Concatenação)

Para concatenar duas strings use a função `strcat()`. Esta função faz parte da biblioteca `string.h`. Sua sintaxe é:

```
char* strcat(char* destino, char* origem)
```

Caso se deseje concatenar apenas um número determinado de caracteres, partindo do início, a função a ser utilizada é a `strncat()`. Sua sintaxe é:

```
char* strncat(char* destino, char* origem, int numrerocaracteres)
```

Exemplo:

```
/* Unindo duas strings usando as
 * funções strcat() e strncat() */

#include <stdafx.h>
#include <stdio.h>
#include <string.h>

void main()
{
    char string1[10], string2[30], string3[30];

    printf("\nConcatenando strings\n");
    printf("-----\n\n");
    printf("Digite string1: ");
    gets(string1);
    printf("\nDigite string2: ");
    gets(string2);
    strcat(string2, string1);
    printf("\nUnindo string1 a string2 : %s", string2);
    strcpy(string3, ""); // inicializa string3 com ""
    strncat(string3, string1, 2);
    strncat(string3, string2, 2);
    printf("\n\nUnindo 2 primeiros caracteres de string1 e string2 a
string3 : %s", string3);
    getchar();
}
```

Comparando duas strings

Para comparar duas strings use a função `strcmp()`. Esta função faz parte da biblioteca `string.h`. Sua sintaxe é:

```
int strcmp(char* str1, char* str2)
```

Se as strings forem iguais a função retorna 0, se `str1` for menor a função retorna um valor menor que 0 e se `str2` for menor a função retorna um valor maior que 0.

Caso se deseje comparar apenas um número determinado de caracteres, partindo do início, a função a ser utilizada é a `strncmp()`. Sua sintaxe é:

```
char* strncmp(char* str1, char* str2, int numrerocaracteres)
```

Há ainda a função `stricmp()`, que realiza a comparação desconsiderando caixa de texto (maiúsculas e minúsculas). Sua sintaxe é:

```
char* stricmp(char* str1, char* str2)
```

Exemplo:

```
/* Comparando strings */

#include <stdafx.h>
#include <stdio.h>
#include <string.h>

void main()
{
    char string1[20],string2[20];
    int retorno;

    printf("\nComparando strings\n");
    printf("-----\n");
    printf("\nEntre com a primeira string: ");
    gets(string1);
    printf("\nEntre com a segunda string: ");
    gets(string2);

    printf("\n");
    retorno = strcmp(string1,string2);
    if(retorno == 0)
        printf("As strings sao iguais.\n");
    else if(retorno < 0)
        printf("A string1 e' menor.\n");
    else
        printf("A string2 e' menor.\n");

    printf("\n");
    retorno = strncmp(string1,string2,2);
    if(retorno == 0)
        printf("Os 2 primeiros caracteres das strings sao iguais.\n");
    else if(retorno < 0)
        printf("A string1 e' menor, comparando apenas os 2 primeiros caracteres.\n");
    else
        printf("A string2 e' menor, comparando apenas os 2 primeiros caracteres\n");

    printf("\n");
    retorno = stricmp(string1,string2);
    if(retorno == 0)
        printf("As strings sao iguais, desconsiderando caixa de texto\n");
}
```

```

        else if(retorno < 0)
            printf("A string1 e' menor, desconsiderando caixa de
texto\n");
        else
            printf("A string2 e' menor, desconsiderando caixa de
texto\n");

        getchar();
    }

```

É muito comum utilizar a estrutura abaixo para testar se as strings são iguais, pois, como o retorno se ocorrer a igualdade é 0, negando-a dará 1, que é `true` no C.

```

if(!strcmp(string1,string2))
    printf("As strings sao iguais.");

```

Convertendo uma string para minúsculas/maiúsculas

Para converter uma string para minúsculas use a função `strlwr()`, e para converter para maiúsculas use a função `strupr()`. Estas funções fazem parte da biblioteca `string.h`. Suas sintaxes são:

```
char* strlwr(char* string)
```

```
char* strupr(char* string)
```

Exemplo:

```

/* Convertendo uma string em minúsculas/maiúsculas */

#include <stdafx.h>
#include <stdio.h>
#include <string.h>

void main()
{
    char string[20];

    printf("\nConvertendo strings\n");
    printf("-----\n");
    printf("\nEntre com uma string: ");
    gets(string);
    strupr(string);
    printf("\nString em maiusculo: %s",string);
    strlwr(string);
    printf("\nString em minusculo: %s",string);
    getchar();
}

```

Localizando a primeira e a última ocorrência de um caracter numa string

Para localizar caracteres em uma string, use a função `strchr()` para localizar a primeira ocorrência e `strrchr()` para localizar a última ocorrência. Estas funções fazem parte da biblioteca `string.h`. Suas sintaxes são:

```
char* strchr(char* string, int character)
```

```
char* strrchr(char* string, int character)
```

Estas funções retornam um ponteiro para a primeira ocorrência de `character`. Caso ele não seja encontrado, a função retornará um ponteiro para o caractere `NULL` (ou `'\0'`) que marca o final da string.

Exemplo:

```
/* Localizando o primeiro e último caracter numa string */

#include <stdafx.h>
#include <stdio.h>
#include <string.h>

void main()
{
    char string[40] = "Teste da funcao strchr e strrchr.";
    char *ptr;

    printf("\n%s\n",string);
    ptr = strchr(string, 's');
    if(*ptr != NULL) // ou if(*ptr != '\0') ou ainda if(*ptr)
        printf("\nA primeira ocorrencia de s e' na posicao %d\n",ptr-
string);
    else
        printf("Caractere nao encontrado.\n");

    printf("\n");
    ptr = strrchr(string, 's');
    if(*ptr != NULL) // ou if(*ptr != '\0') ou ainda if(*ptr)
        printf("A ultima ocorrencia de s e' na posicao %d\n",ptr-
string);
    else
        printf("Caractere nao encontrado.\n");
    getchar();
}
```

Convertendo strings em números

Para converter strings em números utilize as funções abaixo:

Função	Converte strings em
<code>double atof(char* string)</code>	double
<code>int atoi(char* string)</code>	int
<code>long atol(char* string)</code>	long

Estas funções fazem parte da biblioteca `stdlib.h`.

Exemplo:

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

void main()
{
    char string1[20], string2[20];

    printf("\nConvertendo strings em numeros\n");
    printf("-----\n");
    printf("\nEntre com uma string (somente numeros): ");
    gets(string1);
    printf("\nEntre com uma string (numero com casas decimais): ");
    gets(string2);
    int vari = atoi(string1);
    printf("\nstring1 convertido para int = %i", vari);
    long varl = atol(string1);
    printf("\nstring1 convertido para long = %ld", varl);
    double varf = atof(string2);
    printf("\nstring2 convertido para double = %f", varf);
    double soma = atof(string1) + atof(string2);
    printf("\nsoma das conversoes de string1 com string2 = %f", soma);
    getchar();
}
```

Localizando uma substring dentro da string

Para localizar uma substring dentro da string use a função `strstr()`. Esta função faz parte da biblioteca `string.h`. Sua sintaxe é:

```
char* strstr (char* string, char* substring)
```

Se a substring existir dentro da string, a função retornará um ponteiro para a primeira letra da substring, senão retornará `NULL`.

Exemplo:

```
#include <stdafx.h>
#include <stdio.h>
#include <string.h>
```

```

void main()
{
    char string[20], substring[20];
    char *extrai;
    int tamanho;

    printf("\nLocalizando uma substring dentro da string\n");
    printf("-----\n");
    printf("\nEntre com a string: ");
    gets(string);
    printf("\nEntre com a substring: ");
    gets(substring);
    tamanho = strlen(substring);
    extrai = strstr(string, substring);
    printf("\n");
    if(extrai != NULL) // ou if(extrai != '\0') ou ainda if(extrai)
    {
        printf("A string contem a substring.\n");
        printf("A substring comeca na posicao %d.\n", extrai - string);
        printf("A substring tem %d caracteres.\n", tamanho);
    }
    else
        printf("A string nao contem a substring.\n");
    getchar();
}

```

Verificando caracteres

Existem várias funções para manipular caracteres. Algumas delas podem ser vistas a seguir, todas elas definidas na biblioteca `ctype.h`.

Função	O que faz
<code>int toupper(int caracter)</code>	Converte o caracter para maiúsculo.
<code>int tolower(int caracter)</code>	Converte o caracter para minúsculo.
<code>bool isupper(int caracter)</code>	Retorna <code>true</code> (1) se o caracter é maiúsculo e <code>false</code> (0) caso contrário.
<code>bool islower(int caracter)</code>	Retorna <code>true</code> (1) se o caracter é minúsculo e <code>false</code> (0) caso contrário.
<code>bool isalpha(int caracter)</code>	Retorna <code>true</code> (1) se o caracter é uma letra e <code>false</code> (0) caso contrário.
<code>bool isdigit(int caracter)</code>	Retorna <code>true</code> (1) se o caracter é um número e <code>false</code> (0) caso contrário.

Exemplo:

```

#include <stdafx.h>
#include <stdio.h>
#include <ctype.h>

```



```

void main()
{
    char character;

    printf("Digite um character: ");
    character = getchar();
    fflush(stdin);
    printf("\n");
    if (isalpha(character))
        printf("O character e' uma letra %s",isupper(character) ?
"maiúscula." : "minúscula.");
    else if (isdigit(character))
        printf("O character e' um numero.");
    else if (character==' ')
        printf("O character e' um espaco em branco.");
    else
        printf("O character nao e' numero, letra ou espaco.");

    printf("\n\nDigite uma letra minuscula: ");
    character = getchar();
    fflush(stdin);
    character = toupper(character);
    printf("\nLetra convertida para maiuscula: %c",character);
    getchar();
}

```

Exercícios sobre Manipulação de Strings

1. Fazer um programa que, dado um nome completo, informe a abreviatura deste nome. Não se devem abreviar as preposições como: do, de, etc. A abreviatura deve vir separada por pontos. Ex: Paulo Jose de Alma Prado. Abreviatura: P.J.A.P.
2. Fazer um programa que, dadas 2 palavras, determine:
 - a. Se as palavras são iguais.
 - b. Se as palavras são iguais, independente de caixa de texto.
 - c. Caso as palavras sejam diferentes (tanto no a. quanto no b.), qual delas tem maior comprimento.
3. Elaborar um programa que leia uma frase e armazene-a em um vetor de caracteres. Depois, verifique e mostre o número de espaços em branco na frase, o número de vogais, o número de consoantes e o número de dígitos.
4. Fazer um programa que leia uma frase e transfira as consoantes para um vetor e as vogais para outro. Depois mostre cada um dos vetores.
5. Fazer um programa que receba uma string de no máximo 20 caracteres do usuário e armazene em outra string o conteúdo de forma invertida. Lembre-se que o '\0' identifica o final da string. Após, verifique se a string lida é um palíndromo. Palíndromo é a palavra cuja leitura é a mesma, quer se faça da direita para a esquerda, quer da esquerda para a direita. Exemplo: ovo, anilina.

6. Fazer um programa que receba uma string do usuário (máx. 20 caracteres) e um caracter qualquer. O programa deve mostrar a primeira e a última ocorrência do caracter na string.
7. Criar um programa que leia e armazene em um vetor `string` `alunos` o nome de 5 alunos e em uma matriz `notas`, as notas (`double`) das duas provas realizadas e da media dos trabalhos para cada aluno. Após, dentro de um laço, solicitar o nome do aluno que se deseja saber a média final. Comparar com os alunos armazenados e, se encontrar um aluno igual, fazer o cálculo da média e mostrar.

```
MP= (notaprova1*4 + notaprova2*4 + mediatrabalhos*2)/10
```

8. Faça um programa que receba como entradas uma lista de nomes em ordem aleatória e ordene essa lista em ordem alfabética.
9. Faça um programa que leia uma frase de até 60 caracteres e imprima a frase sem os espaços em branco. Imprimir também a quantidade de espaços em branco.
10. Escreva um programa que leia uma palavra e verifique se ela é um palíndromo. Palíndromo é a palavra cuja leitura é a mesma, quer se faça da direita para a esquerda, quer da esquerda para a direita. Exemplo: ovo, anilina, "O TEU DRAMA". Inverso: "AMAR DUETO".
11. Escrever um programa que lê um nome completo em uma string `nome` e joga o que foi digitado para uma outra variável `nome2` de trás pra frente, palavra a palavra, mostrando-a. Exemplo:

```
nome: Paulo Cesar Moreira  
nome2: Moreira Cesar Paulo
```

12. Escrever um programa que lê duas variáveis string e um valor inteiro, que será a posição inicial no qual a segunda string deverá ser inserida na primeira. Mostrar a string final. Exemplo:

```
nome: A casa é grande  
nome2: branca  
posini: 7  
Resultado final: A casa branca é grande
```

Recursividade/Funções Recursivas

Muitas vezes nos deparamos com alguns problemas de computação envolvendo laços de repetição que chegam a ficar muito complexos, e até mesmo difíceis de lidar com comandos específicos para laços, como o `for` ou o `while`. Em muitos desses casos, podemos simplificar estes algoritmos usando a recursividade. Algoritmos recursivos são muito usados em várias áreas, incluindo, por exemplo, a inteligência artificial.

Conceito de recursividade e primeiro exemplo (Fatorial)

A recursividade é um método de programação no qual uma função pode chamar a si mesma, quantas vezes for necessária. As chamadas são colocadas em uma pilha e executadas uma a uma posteriormente. Por exemplo: Imaginemos um programa simples para calcular o fatorial de um número.

```
#include <stdafx.h>

int fatorial(int x)
{
    int i=x;
    if (x==0)
        return 1;
    while (i>1)
    {
        i--;
        x*=i;
    }
    return x;
}

void main()
{
    int num;

    printf("Digite um numero para efetuar o calculo do fatorial: ");
    scanf("%d",&num);
    fflush(stdin);
    printf("\nO fatorial e' %d",fatorial(num));
    getchar();
}
```

Vejam como se calcula o fatorial de um número:

- O fatorial de 0 é, por definição, 1.
- O fatorial de 1 é 1.
- O fatorial de 5 é $5 \times 4 \times 3 \times 2 \times 1$.

Agora, qual é o fatorial de N ? Pela definição, podemos dizer que o fatorial de N é:

- 1, se $N < 2$;
- senão $N * (N-1) * (N-2) \dots * 1$

Vamos a uma versão recursiva da função fatorial:

```
int fatorial(int x)
{
```

```

    if (x==0)
        return 1;
    return x*fatorial(x-1);
    // poderia ser assim:
    // return x==0 ? 1 : x*fatorial(x-1);
}

```

Percebam como foi trocado o bloco de repetição pela chamada à própria função, criando a recursividade.

O `if` testa se o argumento passado para a função é `0` e, em caso afirmativo, a função retorna `1` e acaba. Caso contrário, a função chama a ela própria com o argumento `x` decrementado de uma unidade e multiplica o valor de retorno por `x`. Quando o valor de `x` chegar a `0`, irá retornar `1`, e então, terminará o empilhamento, passando às execuções recursivas, seguindo a ordem da pilha. É muito importante que se tenha uma situação de término da pilha de execução, caso contrário poderá causar uma situação de “stack overflow”, ou seja, um “estouro de pilha”.

Exemplificando, se for chamada a função `fatorial(3)` é criada a seguinte pilha de execução (veja de baixo para cima como foi feito o empilhamento e de cima para baixo como é feita a execução):

```

1*fatorial(0) = 1 (não chama mais a função-condição de parada)
2*fatorial(1) = 2
3*fatorial(2) = 6
4*fatorial(3) = 24 (resultado retornado para a função chamadora)

```

A recursividade é de fato um recurso poderoso em qualquer linguagem de programação que suporte modularização, não ficando exclusivo à linguagem C. Porém, existe um preço a pagar pela recursividade.

No exemplo dado, a função não recursiva será ligeiramente mais rápida que sua versão recursiva devido a dois efeitos da recursividade: empilhamento e fase ociosa.

O empilhamento ocorre sempre que uma função é ativada. A ativação da função provoca os seguintes efeitos:

- Criação das variáveis locais na pilha (os parâmetros são variáveis locais à função e são alocados na pilha).
- Salvamento da pilha atual, para que seja possível o retorno.
- Desvio da execução do programa para a função sendo chamada.

Estas operações são um overhead (processamento ou armazenamento em excesso) para uma função recursiva. O outro efeito é a fase ociosa. A função recursiva pode ser dividida em duas fases:

- Fase ativa, que é quando a função executa algum trabalho.
- Fase ociosa, ou passiva, que é quando está retornando e desempilhando.

Enquanto a fase ativa provoca o empilhamento, a fase passiva provoca o desempilhamento. A função em questão irá ter uma fase ativa do mesmo tamanho da fase ociosa, o que também é um overhead.

Outra desvantagem da função recursiva refere-se ao limite de pilha. A pilha não é uma grandeza ilimitada. Assim sendo, o empilhamento é um recurso que pode ser escasso. O empilhamento exagerado pode levar a um erro do tipo "stack overflow", que é justamente o abuso do uso do empilhamento.

Apesar destas desvantagens, a recursividade é vantajosa para simplificar os algoritmos que têm natureza recursiva. Sempre é possível escrever uma função para resolver o mesmo problema sem recursividade. O que deve ditar a decisão de usar recursividade é:

- Clareza do código e simplicidade do algoritmo
- Penalização mínima de performance. Existem situações em que a performance da função recursiva é melhor que a da função não recursiva.

Segundo exemplo (Série de Fibonacci)

A série de Fibonacci tem como regra que, os dois primeiros termos da série são 1 e 1. A partir deles, os demais termos são construídos pela regra:

$$t_n = t_{n-1} + t_{n-2}$$

Exemplo: Os sete primeiros termos da série são: 1 1 2 3 5 8 13

Vejamos como ficaria a implementação da série de Fibonacci de maneira recursiva:

```
#include <stdio.h>

long fibonacci(int n)
{
    if (n <= 1)
        return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

void main()
{
    int num;

    printf("Digite o número de ordem da série de fibonacci: ");
    scanf("%d", &num);
    fflush(stdin);
```

```

    printf("\nO termo dessa ordem e' %d", fibonacci(num));
    getchar();
}

```

Exemplificando, se for chamada a função `fibonacci(7)` é criada a seguinte pilha de execução (veja de baixo para cima como foi feito o empilhamento e de cima para baixo como é feita a execução):

```

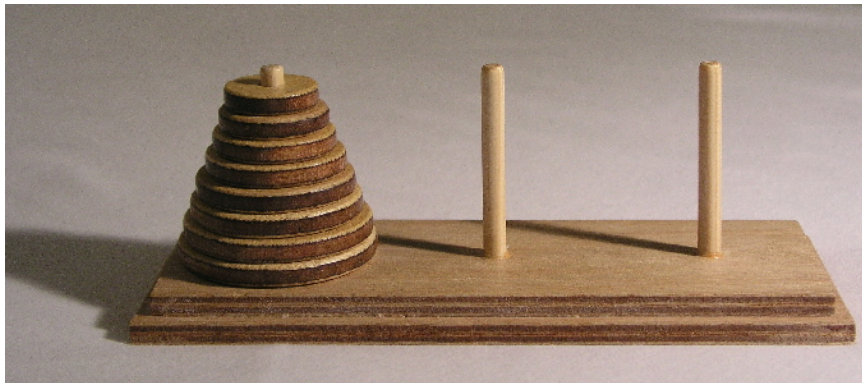
1 (condição de parada)
fibonacci(2) + fibonacci(1) = 2
fibonacci(3) + fibonacci(2) = 3
fibonacci(4) + fibonacci(3) = 5
fibonacci(5) + fibonacci(4) = 8
fibonacci(6) + fibonacci(5) = 13 (resultado final)

```

Torre de Hanói

[Fonte: Wikipedia] As Torres de Hanói são um quebra-cabeça que consiste em uma base contendo três pinos, onde num deles, são dispostos sete discos uns sobre os outros, em ordem crescente de diâmetro, de cima para baixo. O problema consiste em passar todos os discos de um pino para outro qualquer, usando um dos pinos como auxiliar, de maneira que um disco maior nunca fique em cima de outro menor em nenhuma situação. O número de discos pode variar sendo que o mais simples contém apenas três.

As Torres de Hanói tem sido tradicionalmente considerada como um procedimento para avaliação da capacidade de memória de trabalho, e principalmente de planejamento e solução de problemas. A figura a seguir mostra uma Torre de Hanói.



É interessante observar que o número mínimo de "movimentos" para conseguir transferir todos os discos da primeira estaca à terceira é $2^n - 1$, sendo n o número de discos. Logo:

- Para solucionar um Hanói de 3 discos, são necessários $2^3 - 1$ movimentos = 7 movimentos.
- Para solucionar um Hanói de 7 discos, são necessários 127 movimentos.
- Para solucionar um Hanói de 15 discos, são necessários 32.767 movimentos.

- Para solucionar um Hanói de 64 discos, como diz a lenda, são necessários 18.446.744.073.709.551.615 movimentos.

Você pode encontrar uma versão na web de um jogo que simula a Torre de Hanói no site <http://www.ufrgs.br/faced/slomp/hanoi>.

Exercício – Torre de Hanói

1. Fazer um programa que crie uma função recursiva para mostrar os passos necessários para resolver o problema da Torre de Hanói, independente do número de discos, que deve ser informado pelo usuário. A seguir, o descritivo do problema:

O problema da Torre de Hanói e as instruções para obter a solução

O problema consiste numa pilha de n discos e 3 hastes. O jogo funciona assim:

- um disco de tamanho menor não pode estar debaixo de um disco maior.
- podemos retirar apenas o disco superior de uma determinada pilha.

Em primeiro lugar, vamos definir as entradas e saídas. Não é difícil ver que o número de entradas é simplesmente o número n de discos. Já as saídas poderão ser as mais variadas (por exemplo, uma tela de animação). Vamos adotar a saída em modo texto, com instruções no seguinte formato:

Mova o disco 9 da haste X ate a haste X.

Para facilitar as idéias, vamos chamar as hastes de:

- haste A ou haste de origem
- haste B ou haste de destino
- haste C ou haste auxiliar/ajuda

Agora, faça o programa seguindo o algoritmo abaixo:

```
Procedimento Hanói(n, A, B, C)
  se n > 0 então
    Hanoi(n - 1, A, C, B);
    Mover o disco do topo de A para B;
    Hanoi(n - 1, C, B, A);
  fim se
Fim Procedimento

Chamada externa: Hanoi(n, A, B, C)
```

Busca Binária com Recursividade

Pode-se realizar a busca em um elemento em uma lista linear ordenada de duas formas:

- **Busca Seqüencial:** elementos são pesquisados índice a índice, do início ao fim. Problema: o elemento buscado se encontra no final da lista. Imagine realizar uma busca seqüencial em uma Lista Telefônica!
- **Busca Binária:** divide a lista em duas metades, e fazendo os seguintes testes:
 - Se o item for igual ao que está na metade do vetor, ele foi encontrado.
 - Se for menor, realize a divisão novamente na primeira metade.
 - Se for maior, realize a divisão novamente na segunda metade.

Percebe-se, claramente, que a busca binária é uma escolha melhor. Vejamos um exemplo de busca do caracter **R** em um vetor de caracteres ordenado.

1	2	3	4	5	6	7	8	9	10
A	C	E	H	L	M	P	R	T	Z
					↑ I		↑ X		↑ F

Na segunda divisão, onde o início caiu no índice 6 e o fim no 10, encontrou-se o **R** no índice 8.

A seguir uma função de busca binária utilizando recursividade:

```
void busca_binaria(int* vet, int elemento, int inicio, int fim){
    int meio=(inicio+fim)/2;
    if (fim<inicio)
        printf("Elemento nao encontrado");
    else
        if (vet[meio]==elemento)
            printf("Elemento esta na posição %d", meio);
        else
        {
            if (vet[inicio]<elemento)
                inicio=meio+1;
            else
                fim=meio-1;
            busca_binaria(vet, elemento, inicio, fim);
        }
}
```

Exercícios – Busca Binária

1. Fazer um programa que leia um conjunto de 10 valores inteiros e armazene em um vetor. Após, ordene o vetor e, em seguida, leia o valor que o usuário deseja encontrar no vetor. Através da função `busca_binaria` mostre o resultado.
2. Fazer um programa semelhante ao anterior, mas que funcione para caracteres, ou seja, é necessário criar uma função recursiva para encontrar elementos em um vetor de caracteres.

Ordenação Quicksort com Recursividade

Um dos algoritmos mais eficaz para ordenação é o quicksort. A idéia por trás do algoritmo é escolher um valor "médio" (o valor do meio, que pode não corresponder exatamente ao valor do meio do vetor, depois de este estar ordenado) do vetor, passar todos os valores maiores do que ele para a frente e todos os menores para trás.

Ficamos então com o vetor dividido em duas partes, uma tem todos os valores menores que o valor escolhido, a outra tem todos os valores maiores que o valor escolhido. Aplicamos agora o algoritmo a cada uma das partes. O processo é repetido até atingirmos partes de tamanho 1.

O Quicksort adota a estratégia de **dividir e conquistar**, que consiste em dividir um problema maior recursivamente em problemas menores até que o problema possa ser resolvido diretamente. Os passos são:

1. Escolha um elemento da lista, denominado pivô.
2. Rearranje a lista de forma que todos os elementos anteriores ao pivô sejam menores ou iguais a ele, e todos os elementos posteriores ao pivô sejam maiores ou iguais a ele. Ao fim do processo o pivô estará em sua posição final. Essa operação é denominada partição.
3. Recursivamente, ordene a sub-lista dos elementos menores e a sub-lista dos elementos maiores.

A base da recursão são as listas de tamanho zero ou um, que estão sempre ordenadas. O processo é finito, pois a cada iteração pelo menos um elemento é posto em sua posição final e não será mais manipulado na iteração seguinte.

A seguir uma função de ordenação quicksort utilizando recursividade:

```
void ordena_quicksort(int vet[], int inicio, int fim)
{
    if (fim-inicio>0)
    {
        int pivo=vet[inicio];
        int esq=inicio+1;
        int dir=fim;
        int aux;
```

```

while (esq < dir)
{
    if (vet[esq] <= pivo)
        esq++;
    else
    {
        aux = vet[esq];
        vet[esq] = vet[dir];
        vet[dir] = aux;
        dir--;
    }
}
if (vet[esq] > pivo)
    esq--;
aux = vet[inicio];
vet[inicio] = vet[esq];
vet[esq] = aux;
ordena_quicksort(vet, inicio, esq-1);
ordena_quicksort(vet, dir, fim);
}
}

```

Exercício – Ordenação Quicksort

1. Refaça os dois programas sobre busca binária, utilizando a ordenação quicksort para realizar a ordenação.