Linguagem C

Ricardo Terra

rterrabh (at) gmail.com

CV

- **Nome**: Ricardo Terra
- **Email**: rterrabh (at) gmail.com
- **www**: ricardoterra.com.br
- **Twitter**: rterrabh
- **Lattes**: lattes.cnpq.br/ 0162081093970868

Ph.D. (UFMG/UWaterloo),

Post-Ph.D. (INRIA/Université Lille 1)

Background

Acadêmico : UFLA (desde 2014), UFSJ (1 ano), FUMEC (3 anos), UNIPAC (1 ano), FAMINAS (3 anos)

Profissional : DBA Eng. (1 ano), Synos (2 anos), Stefanini (1 ano)

1. Introdução – Conteúdo

Introdução

Tópicos Importantes

Introdução – Tópicos Importantes

- Origem do C
  - Dennis Ritchie
  - BCPL $\rightarrow$ B $\rightarrow$ C em 1970
- Devido à popularidade de microcomputadores, um grande número de implementações surgiram
  - compatíveis, porém com algumas discrepâncias
- Para evitar discrepâncias, o ANSI *(American National Standards Institute)*, em 1983, criou o padrão C ANSI
  - Modificado em 1989 (C89)
  - Modificado em 1999 (C99)
  - Modificado em 2009? Não

Linguagem

- Estruturada
  - exceto por uma única característica
- Médio Nível
  - nível mais alto: Pascal, COBOL, Java...
  - médio nível: C, C++...
  - nível mais baixo: Assembly...
- Para programadores
  - foi criada, influenciada e testada por programadores
  - não se admire que C seja a linguagem mais popular entre excelentes programadores

Compiladores X Interpretadores

- Interpretador lê o código-fonte linha a linha, executando a instrução específica daquela linha
- Compilador lê o programa inteiro, converte-o em um código-objeto (ou código de máquina) de modo que o computador consiga executá-lo diretamente

- *Qual é mais rápido?*

Forma de um programa C

- Declarações globais
- Funções definidas pelo programador
- Função `main`

```
declarações globais

tipo-de-retorno funcao(lista-de-parâmetros) {
    ...
}

...

int main(int argc, char* argv[]) {
    ...
}
```

Exemplo de um programa C

```
1  #include<stdio.h>
   #include<math.h>
3
   const double PI = 3.141596;
5
   double calcularArea(double raio) {
7      return PI * pow(raio, 2);
   }
9
   int main(int argc, char *argv[]) {
11     double raio, area;
       printf("Digite o raio: ");
13     scanf("%lf", &raio);
       area = calcularArea(raio);
15     printf("Area = %lf", area);
       return 0;
17 }
```

Um pouco sobre a função `main`

- A função **`main`** é o ponto de entrada de uma aplicação C. Usualmente, um aplicação C se inicia no método **`main`** e, a partir dele, invoca diversas outras funções. Por fim, essa função retorna um valor inteiro

- Esse valor retornado não é impresso, serve apenas para indicar ao sistema operacional ou para um aplicativo que invocou o seu aplicativo qual foi o código de retorno

- Por padrão, o retorno **`0`** indica que o aplicativo encerrou conforme esperado. Os outros valores podem ser definidos conforme a necessidade de cada aplicação
  - Normalmente, valores negativos se referem a erros

Valores de retorno da função `main`

- Nesta apostila, adota-se os seguintes valores de retorno:
  - `0` → encerramento conforme esperado
  - `-1` → memória insuficiente
  - `-2` → argumentos de linha de comando incorretos
  - `-3` → problema ao abrir arquivo

Palavras reservadas

- auto
- break
- case
- char
- const
- continue
- default
- do
- double
- else
- enum
- extern
- float
- for
- goto
- if
- int
- long
- register
- return
- short
- signed
- sizeof
- static
- struct
- switch
- typedef
- union
- unsigned
- void
- volatile
- while

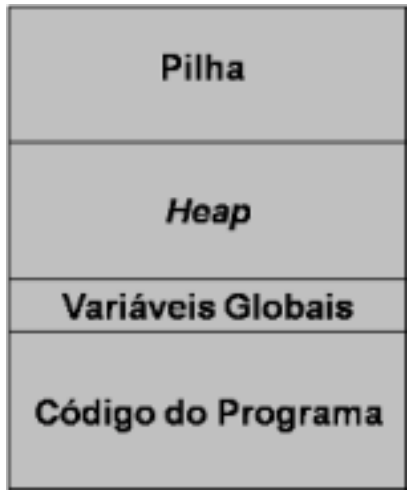
Processo de compilação

- Criar o programa
- Compilar o programa
- Linkeditar o programa com as funções necessárias da biblioteca

- *Por utilizarmos uma IDE de desenvolvimento esse processo fica bem mais transparente*

Mapa de memória em C

- Quatro regiões logicamente distintas:
  - código do programa
  - variáveis globais
  - pilha (*stack*)
  - *heap*
    - região de memória livre a ser requisitada (alocação dinâmica)



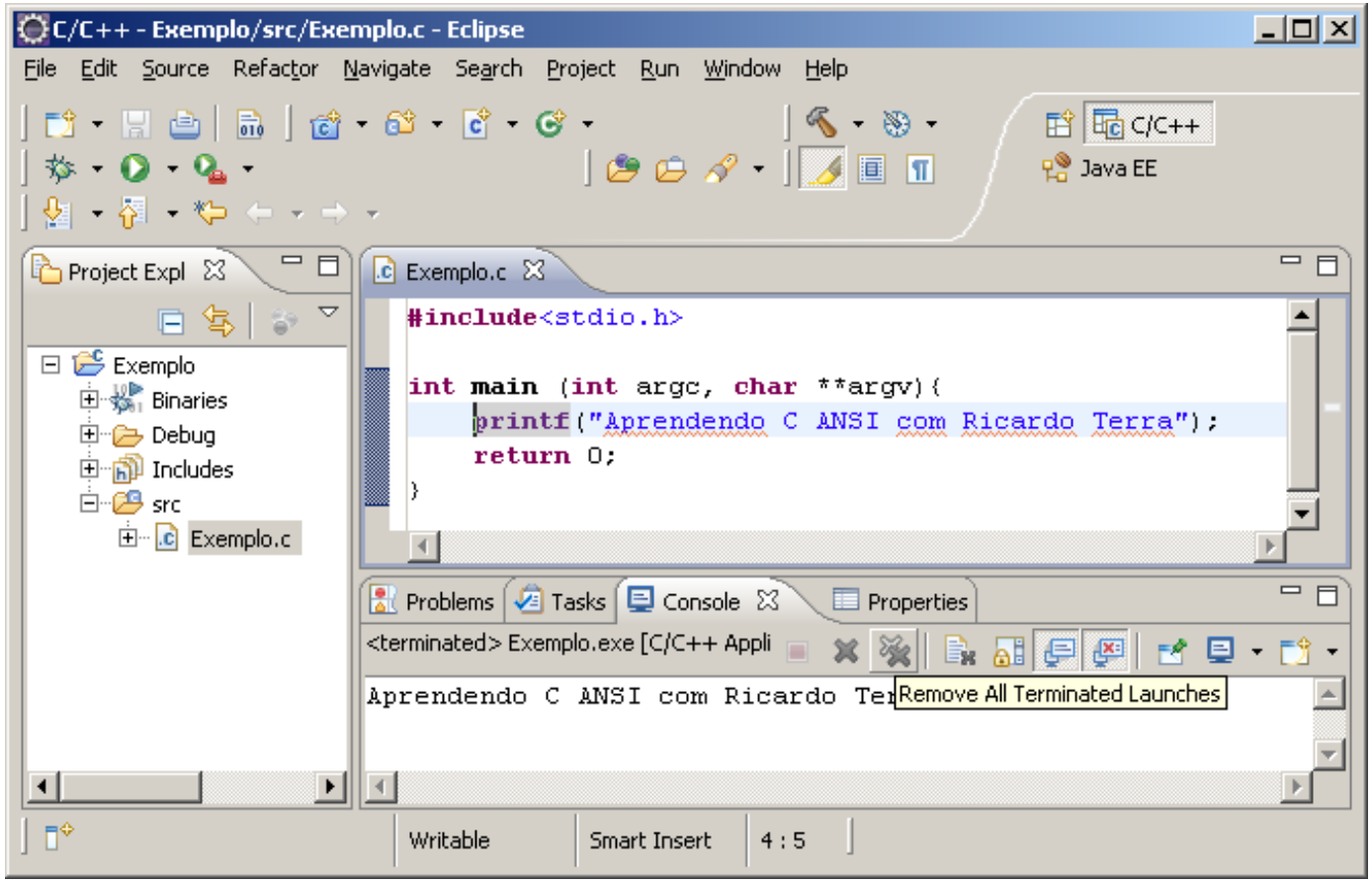
C versus C++

- C++ é uma versão estendida e melhorada da linguagem C que é projetada para suportar programação orientada a objetos
- C++ contém e suporta toda a linguagem C e mais um conjunto de extensões orientadas a objetos
- Portanto:
  - Você não pode programar em C++ se não souber C
  - Tudo que aprender da linguagem C, poderá ser utilizado em C++

- *Compiladores compatíveis?*

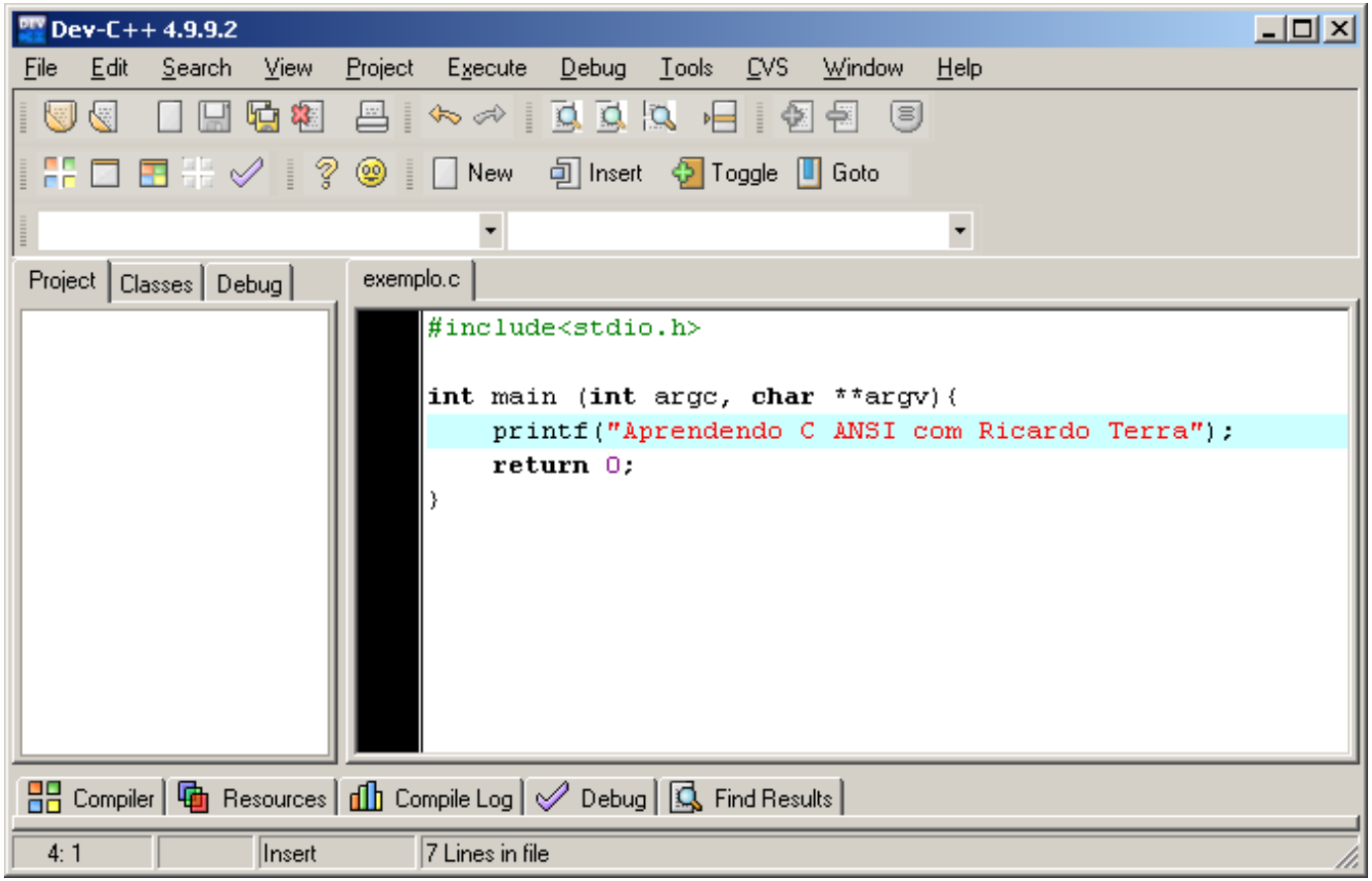
IDEs

- Eclipse (*ver material disponibilizado de como configurar*)



IDEs

- Dev-Cpp 5 (*http://www.bloodshed.net/devcpp.html*)



2. Sintaxe Básica – Conteúdo

Sintaxe Básica

Expressões

Definições

- Expressão é o elemento mais fundamental da linguagem C
- São mais gerais e mais poderosas que na maioria das outras linguagens de programação
- As expressões são formadas por:
  - dados (representados por variáveis ou constantes)
  - operadores
- Exemplos:
  - `2 + 4 * (7 - 3)`
  - `(-b + sqrt(delta)) / (2 * a)`
  - `calcularArea(raio=3.98)`
  - `printf((a%2==0 || ++b<=pow(x,y)) ? "SÍ": "NO")`

Sintaxe Básica – Expressões

- Tipos básicos:
  - Caractere (**`char`**)
  - Inteiro (**`int`**)
  - Ponto flutuante (**`float`**)
  - Ponto flutuante de precisão dupla (**`double`**)
  - Sem valor (**`void`**)
- Modificadores:
  - **`signed`**
  - **`unsigned`**
  - **`long`**
  - **`short`**

Tipo `char`

- Representa um caractere ASCII
- 1 byte
- Na verdade, é um inteiro sem sinal (0 a 255)

Exemplo

```
1  #include<stdio.h>

3  int main(int argc, char *argv[]) {
       char ch;
5      scanf("%c", &ch);
       printf("%c", ch); /*Imprime o caractere lido*/
7
       ch = 'A';
9      printf("%d", ch); /*Imprime o codigo ASCII numerico*/

11     ch = 65;
       printf("%c", ch); /*Imprime o caractere*/
13
       return 0;
15 }
```

Tabela ASCII (parcial)

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040	&#032;	Space	64	40	100	&#064;	@	96	60	140	&#096;	`
1	1	001	Start of Header	33	21	041	&#033;	!	65	41	101	&#065;	A	97	61	141	&#097;	a
2	2	002	Start of Text	34	22	042	&#034;	"	66	42	102	&#066;	B	98	62	142	&#098;	b
3	3	003	End of Text	35	23	043	&#035;	#	67	43	103	&#067;	C	99	63	143	&#099;	c
4	4	004	End of Transmission	36	24	044	&#036;	$	68	44	104	&#068;	D	100	64	144	&#100;	d
5	5	005	Enquiry	37	25	045	&#037;	%	69	45	105	&#069;	E	101	65	145	&#101;	e
6	6	006	Acknowledgment	38	26	046	&#038;	&	70	46	106	&#070;	F	102	66	146	&#102;	f
7	7	007	Bell	39	27	047	&#039;	'	71	47	107	&#071;	G	103	67	147	&#103;	g
8	8	010	Backspace	40	28	050	&#040;	(	72	48	110	&#072;	H	104	68	150	&#104;	h
9	9	011	Horizontal Tab	41	29	051	&#041;	)	73	49	111	&#073;	I	105	69	151	&#105;	i
10	A	012	Line feed	42	2A	052	&#042;	*	74	4A	112	&#074;	J	106	6A	152	&#106;	j
11	B	013	Vertical Tab	43	2B	053	&#043;	+	75	4B	113	&#075;	K	107	6B	153	&#107;	k
12	C	014	Form feed	44	2C	054	&#044;	,	76	4C	114	&#076;	L	108	6C	154	&#108;	l
13	D	015	Carriage return	45	2D	055	&#045;	-	77	4D	115	&#077;	M	109	6D	155	&#109;	m
14	E	016	Shift Out	46	2E	056	&#046;	.	78	4E	116	&#078;	N	110	6E	156	&#110;	n
15	F	017	Shift In	47	2F	057	&#047;	/	79	4F	117	&#079;	O	111	6F	157	&#111;	o
16	10	020	Data Link Escape	48	30	060	&#048;	0	80	50	120	&#080;	P	112	70	160	&#112;	p
17	11	021	Device Control 1	49	31	061	&#049;	1	81	51	121	&#081;	Q	113	71	161	&#113;	q
18	12	022	Device Control 2	50	32	062	&#050;	2	82	52	122	&#082;	R	114	72	162	&#114;	r
19	13	023	Device Control 3	51	33	063	&#051;	3	83	53	123	&#083;	S	115	73	163	&#115;	s
20	14	024	Device Control 4	52	34	064	&#052;	4	84	54	124	&#084;	T	116	74	164	&#116;	t
21	15	025	Negative Ack.	53	35	065	&#053;	5	85	55	125	&#085;	U	117	75	165	&#117;	u
22	16	026	Synchronous idle	54	36	066	&#054;	6	86	56	126	&#086;	V	118	76	166	&#118;	v
23	17	027	End of Trans. Block	55	37	067	&#055;	7	87	57	127	&#087;	W	119	77	167	&#119;	w
24	18	030	Cancel	56	38	070	&#056;	8	88	58	130	&#088;	X	120	78	170	&#120;	x
25	19	031	End of Medium	57	39	071	&#057;	9	89	59	131	&#089;	Y	121	79	171	&#121;	y
26	1A	032	Substitute	58	3A	072	&#058;	:	90	5A	132	&#090;	Z	122	7A	172	&#122;	z
27	1B	033	Escape	59	3B	073	&#059;	;	91	5B	133	&#091;	[	123	7B	173	&#123;	{
28	1C	034	File Separator	60	3C	074	&#060;	<	92	5C	134	&#092;	\	124	7C	174	&#124;	|
29	1D	035	Group Separator	61	3D	075	&#061;	=	93	5D	135	&#093;	]	125	7D	175	&#125;	}
30	1E	036	Record Separator	62	3E	076	&#062;	>	94	5E	136	&#094;	^	126	7E	176	&#126;	~
31	1F	037	Unit Separator	63	3F	077	&#063;	?	95	5F	137	&#095;	_	127	7F	177	&#127;	Del

asciicharstable.com

Tipo `int`

- Representa um inteiro
- 4 bytes (normalmente)

Exemplo

```
1  #include<stdio.h>

3  int main(int argc, char *argv[]) {
       int i;
5      scanf("%d", &i);
       printf("%d", i); /*Imprime o numero lido*/
7
       i = '0';
9      printf("%d", i); /*Imprime o codigo ASCII numerico*/

11     i = 48;
       printf("%c", i); /*Imprime o caractere ASCII correspondente*/
13
       return 0;
15 }
```

Tipo `float`

- Representa um número decimal
- 4 bytes (normalmente)

Exemplo

```
1  #include<stdio.h>

3  int main(int argc, char *argv[]) {
       float f;
5
       scanf("%f", &f);
7      printf("%f", f); /*Imprime o numero lido*/

9      f = 1.234;
       printf("%f", f);
11
       f = 1.3213e2;
13     printf("%.2f", f); /*Imprime com apenas 2 casas decimais*/

15     return 0;
   }
```

Tipo `double`

- Representa um número decimal com maior precisão que `float`
- 8 bytes (normalmente)

Exemplo

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    double d;

    scanf("%lf", &d);
    printf("%lf", d); /*Imprime o numero lido*/

    d = 1.234;
    printf("%lf", d);

    d = 1.3213e2;
    printf("%.2lf", d); /*Imprime com apenas 2 casas decimais*/

    return 0;
}
```

- C não tem um tipo lógico
- Utiliza-se valores inteiros
  - 0 $\rightarrow$ **false**
  - diferente de 0 $\rightarrow$ **true**

Exemplo

```
while (1) {
    /* instruções /*
}
```

Sintaxe Básica – Expressões

- Identificadores
  - nomes de variáveis, funções, rótulos e vários outros objetos definidos pelo usuário
- São case-sensitive
  - Na verdade, C é case-sensitive
- Identificador válido:
  - A primeira letra deve ser uma letra ou um *underline* (_)
  - As letras subsequentes devem ser letras, números ou *underline*
  - Exemplos:
    `idade`, **`nome`**, **`sexo`**, **`telefone1`**, **`a1`**, **`A1`**, **`_x`**, **`_y`**, ...

Sintaxe Básica – Expressões

- Uma variável é uma posição nomeada de memória, que é utilizada para guardar um valor que pode ser modificado pelo programa
- Todas as variáveis devem ser declaradas e inicializadas antes de serem utilizadas (Padrão C89, o qual iremos seguir)
- Declaradas:
  - dentro de funções (*variáveis locais*)
  - como parâmetros de funções (*parâmetros formais*)
  - fora de todas as funções (*variáveis globais*)
- Podem já ser inicializadas

Exemplo de variáveis

```
1  #include<stdio.h>

3  double PI = 3.141596;

5  int soma(int x, int y) {
       return x + y;
7  }

9  int main(int argc, char *argv[]) {
       int c = soma(2, 3);
11     printf("c = %d", c);
       return 0;
13 }
```

Pergunta-se

Quais são as variáveis locais? E as globais? E os parâmetros formais?

Modificadores de tipo de acesso (quantificadores):

- **`const`**
  - O valor da variável não pode ser modificada por seu programa
  - Exemplo: **`const double PI = 3.141596;`**
- **`volative`**
  - O valor da variável pode ser alterada de uma maneira não explicitamente especificada pelo programa
  - Exemplo: **`volative double taxa = 0.65;`**

Especificadores de tipo de classe de armazenamento:

- **`extern`**
  - Diz ao compilador que os tipos e nomes de variável que o seguem foram declarados em outro arquivo fonte
  - Exemplo:
    Já existe um arquivo **`matematica.c`** que possui declarado – como variável global – o valor de **`PI`**. Imagine que esteja desenvolvendo um outro arquivo fonte e queira utilizar o valor do PI já declarado. Você utiliza normalmente, mas deve deixar claro que o valor de **`PI`** está declarado em um outro arquivo.
    Assim declara-se **`PI`** como a seguir: **`extern PI;`**

Especificadores de tipo de classe de armazenamento:

- `static`
  - Variáveis permanentes, i.e., mantém seus valores entre as chamadas
  - Só é inicializado na primeira vez que a função é chamada
  - Funciona como uma variável global (mantendo seu valor), contudo não possui visibilidade global, somente para a própria função (como em variáveis locais)

Exemplo `static`

```
1 #include<stdio.h>

3 int contador() {
      static int count = 0;
5     return ++count;
  }
7
  int main(int argc, char *argv[]) {
9     printf("count = %d\n", contador());
      printf("count = %d\n", contador());
11    printf("count = %d\n", contador());
      return 0;
13 }
```

Pergunta-se

O que será impresso? E se a variável `count` não fosse `static`?

- **`register`**
  - Antigamente: Variável armazenada em um registrador
  - Atualmente: Variável armazenada de tal modo que seja o mais rápido possível
  - Exemplo: **`register int count;`**
- **`auto`**
  - Declaração de variáveis locais, porém não é utilizada
  - Por padrão, todas as variáveis não globais são **`auto`**
  - Exemplo: **`auto int idade;`** $\leftrightarrow$ **`int idade;`**

Em C, constantes referem-se a valores fixos que o programa não pode alterar

- Constantes caracteres: 'A', 'B', 'C', ...
  - São envolvidas por aspas simples (')
- Constantes inteiras: 1, 2, 3, 4, ...
- Constantes em ponto flutuante: 3.14, 4.67, ...

- Constantes Hexadecimais e Octais
  - **Hexadecimal:** inicia com **`0x`**
    - Exemplo: **`0x18`** equivale a **`24`**
  - **Octal:** inicia com **`0`**
    - Exemplo: **`017`** equivale a **`15`**
- Constantes string: "uma string qualquer"
  - Conjunto de caracteres envolvidos por aspas duplas (")

Constantes caractere de barra invertida

- Para a maioria dos caracteres, as aspas simples funcionam bem, porém para *enter*, *tab*, *esc* seria bem complicado representá-los, uma vez que são teclas de ação. Por isto, C criou as constantes especiais de barra invertida

Constantes especiais mais utilizadas

- `\n` **Nova Linha**
- `\r` **Retorno de Carro (CR)**
- `\t` **Tabulação horizontal**
- `\'` **Aspas Simples**
- `\"` **Aspas Duplas**
- `\0` **Nulo**
- `\\` **Barra Invertida**
- `\a` **Alerta**

Quatro classes de operadores:

- Aritméticos
- Relacionais
- Lógicos
- Bit a bit

Operadores Artiméticos

- `+`
- `-`
- `*`
- `/`
- `%` (corresponde ao **`mod`** do Pascal)

Prioridades e Associatividades normais

- Parênteses mudam as prioridades
  - `2 + 2 * 4 != (2 + 2) * 4`
- Operador /
  - Operando inteiros:
    - Divisão Inteira: `(7 / 2 = 3)`
  - Pelo menos um ponto flutuante:
    - Divisão Fracionária:
    - `7.0 / 2.0 = 3.5`
    - `7.0 / 2 = 3.5`
    - `7 / 2.0 = 3.5`
- Operador %
  - Resto de divisão inteira
  - Ex: `4 % 2 = 0` e `5 % 2 = 1`

Operadores Aritméticos em Atribuições Compostas

- Em geral, os operadores aritméticos possuem um operador de atribuição correspondente:
  - Exemplo:
    - `A = A + 2` $\leftrightarrow$ `A += 2`

Outros:

- `-=`
- `*=`
- `/=`
- `%=`

Exemplo de operadores de atribuição compostos

```c
#include<stdio.h>

int main(int argc, char *argv[]) {
    int valor = 10;

    valor += 10;
    printf("%d", valor); /* valor: 20 */

    valor -= 10;
    printf("%d", valor); /* valor: 10 */

    valor *= 5;
    printf("%d", valor); /* valor: 50 */

    valor /= 10;
    printf("%d", valor); /* valor: 5 */

    valor %= 3;
    printf("%d", valor); /* valor: 2 */

    return 0;
}
```

Operadores Aritméticos de incremento e decremento

- `++` (incrementa 1)
- `--` (decrementa 1)

Exemplos:

- `A = A + 1` $\leftrightarrow$ `A += 1` $\leftrightarrow$ `A++` $\leftrightarrow$ `++A`
- `A = A - 1` $\leftrightarrow$ `A -= 1` $\leftrightarrow$ `A--` $\leftrightarrow$ `--A`

Posicionamento (vale para ++ e para --)

- `++` após a variável:
  - pós-incremento (retorna e incrementa)
- `++` antes da variável:
  - pré-incremento (incrementa e retorna)

Exemplos

1.
```
int i = 0;
printf("%d", i++);
printf("%d", i);
printf("%d", --i);
```

2.
```
int i = 0;
printf("%d", ++i);
printf("%d", i);
printf("%d", i--);
printf("%d", i);
```

Pergunta-se

Qual a saída de cada **`printf`**?

```c
#include<stdio.h>

int main(int argc, char *argv[]) {
    int i = 0;
    printf("%d", i++);
    printf("%d", i--);
    printf("%d", --i);
    printf("%d", i);

    i = 6;
    printf("%lf", ++i / 2.0);

    return 0;
}
```

Pergunta-se

Qual a saída de cada **printf**?

Operadores Relacionais:

- `==` (comparação)
- `!=` (diferença)
- `<` `>` `<=` `>=`

Operadores Lógicos:

- São eles: `&` (*and*), `|` (*or*) e `!` (*not*)
- Normalmente, para *and* e *or*, utiliza-se operadores de curto circuito que fazem com que a expressão só seja analisada até que seja possível determinar o resultado
  - São eles: `&&` (*and*) e `||` (*or*)

Operadores Lógicos de Curto Circuito

- Ao encontrarem um valor que determine seu resultado, não testam mais as outras condições, isto é, a expressão só será analisada até que seja possível determinar o resultado

Exemplo: considere que os métodos `a` e `b` retornem booleano

- **`a() & b()`** → executa os dois métodos para obter o resultado
- **`a() | b()`** → executa os dois métodos para obter o resultado
- **`a() && b()`** → se o método **`a`** retornar **`false`**, o resultado será **`false`** e o método **`b`** nem será executado
- **`a() || b()`** → se o método **`a`** retornar **`true`**, o resultado será **`true`** e o método **`b`** nem será executado

Entendeu mesmo?

```
#include<stdio.h>

int a() { return 0; }
int b() { return 1; }
int c() { return 0; }
int d() { return 1; }

int main(int argc, char *argv[]) {
    printf("%d", a() & b() & c() & d());
    printf("%d", a() && b() && c() && d());
    printf("%d", a() || b() || c() || d());

    return 0;
}
```

Pergunta-se

Quais métodos serão executados e qual o retorno dos **`printf`**'s das linhas 9, 10 e 11?

Tiro no pé!

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    int a = 0;
    if (a != 0 & 2 / a > 5) {
        printf("OK");
    } else {
        printf("NOK");
    }

    return 0;
}
```

Pergunta-se

O programa acima apresenta um erro fatal. Como evitá-lo?

Operadores bit a bit:

- `&` (*and*)
  - Ex: `3 & 1` $\leftrightarrow$ `0011 & 0001` $\leftrightarrow$ `0001` $\leftrightarrow$ `1`
- `|` (*or*)
  - Ex: `3 | 1` $\leftrightarrow$ `0011 | 0001` $\leftrightarrow$ `0011` $\leftrightarrow$ `3`
- `^` (*xor*)
  - Ex: `3 ^ 1` $\leftrightarrow$ `0011 ^ 0001` $\leftrightarrow$ `0010` $\leftrightarrow$ `2`

Operadores bit a bit:

- `~` (*complemento de um*)
  - Ex: `3` $\leftrightarrow$ `~0011` $\leftrightarrow$ `1100` $\leftrightarrow$ `12`
- `<<` (*deslocamento à esquerda*)
  - Ex: `4 << 1` $\leftrightarrow$ `0100 << 1` $\leftrightarrow$ `1000` $\leftrightarrow$ `8`
- `>>` (*deslocamento à direita*)
  - Ex: `4 >> 1` $\leftrightarrow$ `0100 >> 1` $\leftrightarrow$ `0010` $\leftrightarrow$ `2`
- `>>>` (*deslocamento à direita com sinal*)
  - Ex: `-127 >>> 1` $\leftrightarrow$ `2147483584`

Exercícios

- `10 & 13` ?
- `10 | 13` ?
- `10 ^ 13` ?
- `0` ?
- `8 >> 1` ?
- `2 << 2` ?

Comentários

- Algumas vezes se deseja explicar um código, explicitar uma informação em linguagem natural. Para isso existem os comentários. Eles não deixam o programa mais lento e seu uso é altamente recomendável.
  Existem dois tipos de comentários em C ANSI:

- Comentário até final de linha. Tudo escrito após ele na linha, é considerado comentário. Exemplo:

```
int rg; //só os números serão armazenados
```

- Comentário de bloco. Tudo escrito entre **/*** e ***/** é considerado comentário. Exemplo:

```
/*
   pode incluir
   várias linhas
*/
```

Sintaxe Básica

Comandos condicionais

Comandos condicionais (`if`, `switch` e operador ternário)

- Comando condicionais são aqueles que dependendo de uma condição executam um bloco, caso a condição não seja atendida, o bloco não será executado
- C provê suporte a três comandos condicionais:
  - **`if`**
  - **`switch`**
  - operador ternário (**`? :`**)

Comandos condicionais - `if`

- Sintaxe do comando **`if`**

```
if ( condicao ) {
    comando1;
    comando2;
    comando3;
}
```

Observações:

- Os parênteses que envolvem a condição são **OBRIGATÓRIOS**
- Diferentemente de Pascal, Delphi (Object Pascal) etc
- A condição deverá retornar um tipo booleano
- Os comandos somente serão executados se a condição for verdadeira

Comandos condicionais - `if`

- O uso dos braços **NÃO** é obrigatório caso seja apenas um único comando

- Porém, a boa prática recomenda a utilização de braços independente do número de comandos
  - Melhor indentação do código

```
if ( 1 ) comando;
```

equivale a:

```
if ( 1 ) {
    comando;
}
```

Comandos condicionais - `if`

- Como faço o conhecido:

  se *verdade* então . . . senão . . .

```
if ( condicao ) {
    comando1;
    comando2;
    comando3;
} else {
    comando4;
    comando5;
    comando6;
}
```

Pergunta-se

Quais comandos serão executados se a condição for verdadeira? E se for falsa?

Comandos condicionais - `if`

```
1 #include<stdio.h>

3 int main(int argc, char *argv[]) {
      int a, b;
5     scanf("%d %d", &a, &b);
      if (a >= b) {
7         printf("%d e maior ou igual a %d!", a, b);
      } else {
9         printf("%d e menor que %d!", a, b);
      }
11    return 0;
  }
```

Comandos condicionais - `if`

- Comando **`if`** podem ser aninhados

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    int nota;
    scanf("%d", &nota);

    if (nota >= 90) {
        printf("Nota A");
    } else if (nota >= 80) {
        printf("Nota B");
    } else if (nota >= 70) {
        printf("Nota C");
    } else {
        printf("Reprovado");
    }
    return 0;
}
```

Exemplo `if`

```
1  #include<stdio.h>

3  int main(int argc, char *argv[]) {
       char op;
5      printf("Digite um operador (+-/*): ");
       scanf("%c", &op);
7
       if (op == '+' || op == '-') {
9          printf("Operador de Baixa Prioridade.\n");
       } else if (op == '/' || op == '*') {
11         printf("Operador de Alta Prioridade.\n");
       } else {
13         printf("Caractere Invalido!\n");
       }
15
       return 0;
17 }
```

Comandos condicionais - `switch`

- Na instrução **`switch`**, uma variável de tipo primitivo **`char`** ou **`int`** é comparada com cada valor em questão. Se um valor coincidente é achado, a instrução (ou instruções) **depois do teste** é executada

Sintaxe:

```
switch( variavel ){
   case 1: comandoA; break;
   case 2: comandoB; break;
   case 3: comandoC; break;
   default: comandoPadrao;
}
```

Comandos condicionais - `switch`

- Se nenhum valor for encontrado, a instrução **`default`** é executada
- O comando **`break`** é necessário para quebrar o **`switch`**, pois assim que encontrada a opção correta, é executado tudo em seguida

Comandos condicionais - `switch`

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    char letra = 'B';
    switch (letra) {
        case 'A': printf("Entrou em A");break;
        case 'B': printf("Entrou em B");
        case 'C': printf("Entrou em C");break;
        case 'D': printf("Entrou em D");break;
        default:  printf("Entrou em Default");
    }
    return 0;
}
```

Pergunta-se

- Qual a saída?
- E se o valor de letra fosse **'C'**? E se fosse **'a'**?

Comandos condicionais - `switch`

```
1 #include<stdio.h>

3 int main(int argc, char *argv[]) {
      char letra = 'B';
5     switch (letra) {
          default:  printf("Entrou em Default");
7         case 'A': printf("Entrou em A");break;
          case 'B': printf("Entrou em B");break;
9         case 'C': printf("Entrou em C");break;
          case 'D': printf("Entrou em D");break;
11    }
      return 0;
13 }
```

Pergunta-se

- Qual a saída?

Comandos condicionais - `switch`

```
1 #include<stdio.h>

3 int main(int argc, char *argv[]) {
      char letra = 'b';
5     switch (letra) {
          case 'A': printf("Entrou em A");break;
7         case 'B': printf("Entrou em B");break;
          default:  printf("Entrou em Default");
9         case 'C': printf("Entrou em C");break;
          case 'D': printf("Entrou em D");break;
11    }
      return 0;
13 }
```

Pergunta-se

- Qual a saída?

Exemplo `switch`

```
1  #include<stdio.h>

3  int main(int argc, char *argv[]) {
       char op;
5      printf("Digite um operador (+-/*): ");
       scanf("%c", &op);
7
       switch (op) {
9          case '+':
           case '-': printf("Operador de Baixa Prioridade.\n");break;
11         case '*':
           case '/': printf("Operador de Alta Prioridade.\n");break;
13         default: printf("Caractere Invalido!\n");
       }
15
       return 0;
17 }
```

Comandos condicionais - Operador Ternário

- Existem situações cujo uso do **`if`** não é "elegante"
- Por exemplo, suponha que a função **`max`** retorne o maior número dentre os dois passados via parâmetros formais

```
1 int max(int a, int b) {
      if (a > b) {
3         return a;
      } else {
5         return b;
      }
7 }
```

Comandos condicionais - Operador Ternário

- O operador ternário é uma expressão, significando que ele devolve um valor
- O operador ternário é muito útil para condicionais (curtas e simples) e tem o seguinte formato:

```
variável = <condição> ?  seTrue :  seFalse;
```

  - A condição pode estar envolvida entre parênteses para facilitar a leitura, contudo não é obrigatório

Exemplos:

```
int a = 2;
int b = 3;
int c = ( a > b ) ? a : b;
```

- Indica que se **a** for maior que **b**, **c** recebe o valor de **a**, caso contrário recebe o valor de **a**, isto é, **c** recebe o maior valor entre **a** e **b**

Pergunta-se

Qual o valor das variáveis abaixo:

```
int peso = ( 2 != 2 ) ? 80 : 63;

char letra = ( 1 == 1 ) ? 'R' : 'T';
```

Comandos condicionais - Operador Ternário

- Não necessariamente o retorno do operador ternário deve ser atribuído a uma variável
- Por exemplo, seu retorno pode ser o retorno de uma função como faz a função **max** com operador ternário:

```
int max(int a, int b) {
    return (a > b) ? a : b;
}
```

- Ou mesmo o retorno pode servir como parâmetro de chamada de uma função:

```
printf((a > b) ? "a maior!" : "b maior!");
printf("%d", (a > b) ? a : b);
```

Sintaxe Básica

Comandos de repetição

Comandos repetição (`while`, `do...while` e `for`)

- Comandos de repetição são utilizados para repetir um bloco de código
- C provê suporte a três comandos de repetição:
  - **`while`** (enquanto)
  - **`do...while`** (faça...enquanto)
  - **`for`** (para)

Comandos repetição - `while`

- O comando **`while`** é utilizado para repetir um bloco de acordo com uma condição
- É considerado um *loop* de pré-teste
  - Isto é, testa a condição antes de executar o bloco

Sintaxe:

```
while ( condicao ) {
    comando1;
    comando2;
    comandoN;
}
```

Comandos repetição - `while`

```
1 #include<stdio.h>

3 int main(int argc, char *argv[]) {
      int i = 0;
5
      while (i < 10) {
7         printf("%d", ++i);
      }
9
      return 0;
11 }
```

Pergunta-se

Qual a saída?

Comandos repetição - `do...while`

- O comando **`do...while`** é semelhante ao **`while`**, contudo é um comando de repetição de pós-teste
  - Isto é, somente ao final da execução do bloco que se verifica a condição

- Geralmente, é utilizado quando se deseja testar a condição somente a partir da segunda iteração
  - Por exemplo, uma leitura da opção de um menu. Pede para digitar uma primeira vez. **Somente** se não digitar uma opção válida que pede para digitar novamente

Sintaxe:

```
do {
    comando1;
    comando2;
    comandoN;
} while ( condicao );
```

Observe o ponto-e-vírgula após os parênteses da condição. Não o esqueça!

Comandos repetição - do...while

```
1 #include<stdio.h>

3 int main(int argc, char *argv[]) {
      int i = 0;
5
      do {
7         printf("%d", ++i);
      } while (1 != 1);
9
      return 0;
11 }
```

Pergunta-se

Qual a saída?

Exemplo

```
1 #include<stdio.h>

3 int main(int argc, char *argv[]) {
      int i;
5
      do {
7         printf("Digite um numero entre 0 e 10: ");
          scanf("%d", &i);
9     } while (i < 0 || i > 10);

11    printf("Numero digitado: %d\n", i);

13    return 0;
  }
```

Pergunta-se

Qual a saída?

Comandos de repetição - `for`

- Comando de repetição mais poderoso da linguagem C
- É composta por:
  - Inicialização: executado uma única vez no início do *loop*
  - Condição: executado sempre antes de cada iteração. Se verdadeira, o bloco é executado. Se falsa, é finalizado
  - Operação : executado sempre ao término de cada iteração

Sintaxe

```
for ( inicializacao ; condicao ; operacao ) {
   comando1;
   comando2;
   ...
   comandoN;
}
```

Exemplo

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    int i;

    for (i = 0; i < 10; i++) {
        printf("%d", i);
    }

    return 0;
}
```

Pergunta-se

Qual a saída?

Sintaxe

- No laço **for**, a *inicialização*, *condição* e *operação* são todas opcionais

Exemplo

```
1 #include<stdio.h>

3 int main(int argc, char *argv[]) {
      int i = 0;
5     for (;;) { /* Sem condicao, admite-se sempre verdade */
          printf("%d", ++i);
7     }
      return 0;
9 }
```

Pergunta-se

Qual a saída?

Comandos de repetição - `for`

- Podemos ter um **`for`** dentro de outro, e outro dentro de outro, e outro dentro de outro...

```
1 #include<stdio.h>

3 int main(int argc, char *argv[]) {
      int i, j;
5
      for (i = 0; i <= 2; i++) {
7         for (j = 0; j < 2; j++) {
              printf("%d %d", i, j);
9         }
      }
11    return 0;
  }
```

Pergunta-se

Qual a saída?

Comandos de repetição - `for`

- Um comando **`for`** pode ter várias inicializações, uma condição complexa e várias operações

Exemplo

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    int i, d;

    for (i = 1, d = 2 * i; i <= 10 || d == 22; i++, d = i * 2) {
        printf("%d - %d\n", i, d);
    }

    return 0;
}
```

Comando `break`

- Inserido dentro de um bloco de repetição (pode também ser **`while`** ou **`do...while`**)
- Caso seja executado, o bloco de repetição é finalizado

Exemplo

```
1 int main(int argc, char *argv[]) {
      int i = 0;
3     for (; i < 10; i++) {
          if (i == 3) {
5             break;
          }
7         printf("%d", i);
      }
9     return 0;
  }
```

Comando `continue`

- Inserido dentro de um bloco de repetição
- Caso seja executado, a iteração atual do bloco de repetição é interrompida e parte para a próxima iteração

Exemplo

```
int main(int argc, char *argv[]) {
    int i;
    for (i = 0; i < 10; i++) {
        if (i == 3 || i == 5) {
            continue;
        }
        printf("%d", i);
    }
    return 0;
}
```

Comando de repetição - `for`

- Logicamente, pode-se utilizar **`break`** e **`continue`** conjuntamente

Exemplo

```
int main(int argc, char *argv[]) {
    int i, j;
    for (i = 0; i < 3; i++) {
        if (i == 1) {
            continue;
        }
        for (j = 0; j < 2; j++) {
            printf("%d %d\n", i, j);
            break;
        }
    }
    return 0;
}
```

Comando de repetição rotulados

- Ao contrário de linguagens como Java, C ANSI não possui blocos de repetição rotulados
- Por exemplo, dependendo uma certa condição de um bloco mais interno, você quer quebrar ou continuar a partir do bloco mais externo

Exemplo em Java

```
1 externo:
  for (i=0; i < 3; i++) {
3     for (j=0; j < 3; j++) {
          if (i == 1 && j == 1){
5             break externo;
          }
7         System.out.printf("%d %d\n", i, j);
      }
9 }
```

Exemplo

- Neste exemplo, se o valor de `j` e de `k` forem `1`, o bloco de repetição de `k` é encerrado e volta a iteração do bloco de repetição de `j`
- E se fosse para encerrar o **`for`** mais externo?

Exemplo

```
1  int main(int argc, char *argv[]) {
       int i, j, k;
3      for (i = 0; i < 3; i++) {
           for (j = 0; j < 3; j++) {
5              for (k = 0; k < 3; k++) {
                   if (j == 1 && k == 1) {
7                      break;
                   }
9                  printf("%d %d %d\n", i, j, k);
               }
11         }
       }
13     printf("Programa Encerrado.");
       return 0;
15 }
```

Solução

- A solução seria utilizar rótulos (*labels*)
- Contudo, seu uso é perigoso e pouco recomendado

Exemplo

```
1  int main(int argc, char *argv[]) {
       int i, j, k;
3      for (i=0; i<3; i++) {
           for (j=0; j<3; j++) {
5              for (k=0; k<3; k++) {
                   if (j == 1 && k == 1){
7                      goto fora;
                   }
9                  printf("%d %d %d\n", i, j, k);
               }
11         }
       }
13     fora:
       printf("Programa Encerrado.");
15     return 0;
   }
```

Sintaxe Básica

Arranjos

Arranjos - Conceito

- Arranjos – também conhecidos como vetor, *array* etc – são coleções de um mesmo tipo em sequência
- Arranjos podem ser de tipos primitivos (`char`, `int`, `float`, `double`), de um outro arranjo, de uma estrutura, de enumeração...
- Pode se ter um arranjo de inteiros ou um arranjo de caracteres ou um arranjo de arranjo de pontos flutuantes
  - Contudo, não se pode ter um arranjo que contenha inteiros e pontos flutuantes

Arranjos - Declaração

- Declaração:

```
int notas[5]; /* Arranjo de 5 inteiros */
char letras[5]; /* Arranjo de 5 caracteres */
```

- Assim como variáveis comuns, os elementos do arranjo **não** são inicializados automaticamente. Contudo, você pode declarar já inicializando:

```
int notas[5] = {4,6,6,9,8};
char letras[5] = {65, 'E', 'I', 'O', 'U'};
char nome[100] = "ANA"; /* string de 100 caracteres */
```

Arranjos - Declaração

- Um arranjo de tamanho `n`, tem suas posições indexadas de `0` a `n-1`. Exemplo:

```
1 int notas[5] = {8,7,8,9,3};
  notas[3] = 7;
3 printf("%d",notas[4]);
```

- Para obter o tamanho de um arranjo, basta dividir o seu tamanho pelo tamanho ocupado por cada elemento utilizando a função `sizeof`. Exemplo:

```
1 int tam = sizeof(notas) / sizeof(int);
```

Declarando, inicializando e iterando um arranjo de inteiros

```
1  #include<stdio.h>

3  int main(int argc, char *argv[]) {
       int notas[5], i;
5
       int tam = sizeof(notas) / sizeof(int);
7
       for (i = 0; i < tam; i++) {
9          notas[i] = 0;
       }
11
       return 0;
13 }
```

Declarando, inicializando e iterando um arranjo de caracteres

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    int i;
    char vogais[5] = { 'A', 'E', 'I', 'O', 'U' };

    for (i = 0; i < sizeof(vogais) / sizeof(char); i++) {
        printf("%c", vogais[i]);
    }

    return 0;
}
```

Arranjos - Acesso aos elementos

- Arranjos permite recuperar ou alterar qualquer um de seus elementos
- Os arranjos em C sempre iniciam-se na posição 0
  - Isto indica que ele termina em uma posição inferior ao tamanho (`n-1`)

Exemplo

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    char letras[10];

    letras[0] = 'A';          /* Atribui a 1a posicao do arranjo */
    printf("%c", letras[0]); /* Imprime a 1a posicao do arranjo */

    letras[1] = 'B';
    letras[2] = 'C';
    /*...*/
    letras[9] = 'H';
    /* letras[10] = 'I'; --> ERRO */

    return 0;
}
```

- O erro acima acessa uma área de memória não reservada ao arranjo. Isso pode alterar o funcionamento normal do programa ou até mesmo "derrubá-lo"

Entendendo arranjos de caracteres

- Um arranjo de caracteres é como qualquer outro arranjo. Se ele tem tamanho `n`, ele tem `n` posições disponíveis indexadas de `0` a `n-1`
- Um arranjo de caracteres não é um *string*
- Em C, não existe o tipo *string*. Ele é simulado por um arranjo de caracteres em que o caractere `'\0'` delimita o seu final

Arranjo de caracteres como Arranjo de caracteres

- Suas posições válidas são – como de todos os arranjos – de `0` a `n-1`
- Seu tamanho é calculado pelo uso da função `sizeof`
- A leitura/impressão de cada caractere usa-se `%c`

Exemplo

```
1  #include<stdio.h>

3  int main(int argc, char *argv[]) {
       char placa[7] = { 'B', 'A', 'D', '0', '0', '0', '7' };
5      int i;

7      for (i = 0; i < sizeof(placa) / sizeof(char); i++) {
           printf("%c", placa[i]);
9      }

11     return 0;
   }
```

Arranjo de caracteres como *string*

- Suas posições válidas são de `0` a `n-2`. No pior caso, a posição `n-1` deve conter o terminador nulo
- Seu tamanho é calculado pelo uso da função `strlen` da biblioteca `string.h`
- A leitura utiliza a função `gets` e a impressão usa-se `%s`

Exemplo

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

int main(int argc, char *argv[]) {
    char nome[100];

    gets(nome); /* Le string */

    printf("O nome digitado foi %s (%d caracteres)", nome, (int) strlen(nome));

    return 0;
}
```

Arranjos - Cópia

- Observe o seguinte trecho:

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

- Cópia completamente errada:
  - vB = vA;

- Cópia correta:

```
for (i=0; i<4; i++){
    vB[i] = vA[i];
}
```

  - Isto é, uma posição de cada vez

- No **for**, utilizei a literal 4 para limitar a iteração somente para demonstração, contudo qual seria mais apropriado?

Arranjos Multidimensionais

- Pode-se criar um arranjo de arranjos
- O mais comum é o bidimensional que vemos como uma matriz
- A declaração é somente acrescentar o número de colunas
- Por exemplo: `int matriz[4][3]` declara-se uma matriz de 4 linhas e 3 colunas

```
matriz[4][3] = {{1,0,0},{0,1,2},{2,3,4},{0,6,7}};
```

Representação:

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 2 \\ 2 & 3 & 4 \\ 0 & 6 & 7 \end{matrix}$$

Arranjo Bidimensional - Exemplo

- Criando uma matriz 3x2, zerando e setando valores

```
1  #include<stdio.h>

3  int main(int argc, char *argv[]) {
       int matriz[3][2], i, j; /* Declara uma matriz 3x2 */
5      for (i = 0; i < 3; i++) {
           for (j = 0; j < 2; j++) {
7              matriz[i][j] = 0;
           }
9      }

11     matriz[0][0] = 1; /* Atribui o valor 1 ao canto superior esquerdo */
       matriz[2][1] = 7; /* Atribui o valor 7 ao canto inferior direito */
13
       return 0;
15 }
```

matriz

1	0
0	0
0	7

Arranjo Bidimensional - Exemplo com `sizeof`

- O exemplo anterior não utilizou `sizeof` porque o cálculo do tamanho depende do entendimento de ponteiros. Só para saberem, o certo seria:

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    int matriz[3][2], i, j; /* Declara uma matriz 3x2 */
    for (i = 0; i < sizeof(matriz) / sizeof(matriz[i]); i++) {
        for (j = 0; j < sizeof(matriz[i]) / sizeof(int); j++) {
            matriz[i][j] = 0;
        }
    }

    matriz[0][0] = 1; /* Atribui o valor 1 ao canto superior esquerdo */
    matriz[2][1] = 7; /* Atribui o valor 7 ao canto inferior direito */

    return 0;
}
```

Exemplo Completo

- Leitura de cada elemento de uma matriz 2x3 e posterior impressão

```c
#include<stdio.h>

int main(int argc, char *argv[]) {
    int matriz[2][3], i, j;

    for (i = 0; i < sizeof(matriz) / sizeof(matriz[i]); i++) {
        for (j = 0; j < sizeof(matriz[i]) / sizeof(int); j++) {
            printf("Digite matriz[%d][%d]: ", i, j);
            scanf("%d", &matriz[i][j]);
        }
    }

    for (i = 0; i < sizeof(matriz) / sizeof(matriz[i]); i++) {
        for (j = 0; j < sizeof(matriz[i]) / sizeof(int); j++) {
            printf("%d\t", matriz[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Sintaxe Básica

Variáveis

Variáveis

- Uma variável é uma posição nomeada de memória

- São declaradas basicamente em três lugares:
  - dentro de funções → ***variáveis locais***
  - nos parâmetros de funções → ***parâmetros formais***
  - fora de todas as funções → ***variáveis globais***

Variáveis

- **Variáveis locais**
  - são reconhecidas apenas dentro do bloco em que foi declarada e existem apenas enquanto o bloco de código em que foi declarada está sendo executado
- **Parâmetros formais**
  - serão vistos quando abordarmos funções, mas, para adiantar, se comportam exatamente como variáveis locais que são reconhecidas apenas dentro da função
- **Variáveis globais**
  - são variáveis declaradas fora de qualquer função que são reconhecidas por todo o programa e podem ser utilizadas por qualquer bloco de código

Exemplo

```
1  #include<stdio.h>
   #include<math.h>
3
   const double PI = 3.141596; /* Variavel Global */
5
   double area(double r) { /* Parametro Formal */
7      return PI * pow(r, 2);
   }
9
   int main(int argc, char *argv[]) {
11     double raio; /* Variavel Local */
       printf("Digite o raio: ");
13     scanf("%lf", &raio);
       printf("A area da circunferencia e: %lf", area(raio));
15     return 0;
   }
```

Variáveis Globais

- Características:
  - São declaradas no arquivo fonte fora do escopo de qualquer função
  - Existem durante todo o ciclo de vida do programa
  - Só são acessíveis às funções declaradas depois delas no mesmo arquivo fonte

Variáveis Globais - Exemplo

```
#include<stdio.h>

void func1() {
    printf("Var. Global g nao acessivel");
}

int g; /* variavel global */

void func2() {
    g++;
    printf("Var. Global g acessivel: %d", g);
}

int main(int argc, char *argv[]) {
    g = 5;
    func1();
    func2();
    return 0;
}
```

Variáveis

- Uma variável dentro de um bloco interno esconde a variável de mesmo nome no bloco externo
- Observe os exemplos a seguir em que temos variáveis com o mesmo nome em níveis lexicográficos diferentes e tente aferir a saída gerada pelo programa

Variáveis - Níveis Lexicográficos

```
1 #include<stdio.h>

3 int i = 8; /*variavel global*/

5 int main(int argc, char *argv[]) {
      int i = 3;
7     printf("%d", i);
      return 0;
9 }
```

Variáveis - Níveis Lexicográficos

```
1  #include<stdio.h>

3  int a; /* variavel global */

5  void func(int a) { /* parametro formal */
       printf("%d", a);
7      if (a > 2) {
           int a = 7; /* variavel local */
9          printf("%d", a);
       }
11     a += 7;
       printf("%d", a);
13 }

15 int main(int argc, char *argv[]) {
       a = 3; /* atribuicao a variavel global */
17     func(a);
       printf("%d", a);
19     return 0;
   }
```

Sintaxe Básica

Funções

Função

- Um programa C é uma coleção de funções
- Uma das funções deve se chamar **`main`**
  - ponto de entrada
- Uma função pode:
  - receber parâmetros
  - declarar variáveis locais
  - conter instruções executáveis
  - retornar um valor

Função

- Uma função não pode declarar outra função
  - Este é o motivo pelo qual a linguagem C não é classificada integralmente como estruturada em blocos

Isso não pode!

```
int f() {
    ...
    ...
    int g() { /* Nao pode! */

    }
    ...
    ...
}
```

Função

- A função deve ser declarada antes do local onde é chamada
- O comando **`return`** efetua o retorno (término) da função
- Uma função pode ou não ter um retorno
- Uma função pode ou não ter parâmetros formais

Exemplos

```
1 void m1() { ... } /* sem retorno e sem parametros formais */

3 void m2(double x) { ... } /* sem retorno e com um par. formal */

5 int m3() { ... } /* com retorno e sem parametros formais */

7 int m4(char c, int f) { ... } /* com retorno e com dois param. */
```

Ordem de Declaração

- Função deve ser declarada antes de seu uso: no caso, **soma** está declarada antes de **main**

```
1  #include<stdio.h>

3  int soma(int x, int y) { /* parametros Formais */
       return x + y;
5  }

7  int main(int argc, char *argv[]) {
       int a = 2, b = 3, total;
9
       /* chamada a funcao: valores de a e b copiados para x e y */
11     total = soma(a, b);

13     printf("Soma: %d", total);
       return 0;
15 }
```

Função

- **Sintaxe:**

```
retorno nome ( < param {, param} > ) { corpo }
```

Exemplos

```
1 void imprimir() { ... }

3 int dobro(int x) { ... }

5 double somar(double a, double b) { ... }

7 void listar(int notas[], int tam) { ... }
```

- Convém salientar que a passagem de parâmetros em C é sempre por valor, i.e., os valores são copiados da origem para o destino

Exemplo - Passagem por Valor

```
1  #include<stdio.h>

3  void func(int k) {
       k = 3;
5  }

7  int main(int argc, char *argv[]) {
       int i = 2;
9      func(i);
       printf("%d", i);
11     return 0;
   }
```

Pergunta-se

Qual o valor de `i` impresso?

Função

- Retorno de funções
  - Uma função pode retornar valores de qualquer tipo, exceto arranjos ou outras funções
  - Uma função que retorna nada, deve ter seu retorno declarado como **`void`**
  - A expressão que segue o **`return`** é o valor retornado
    - não se deve colocar parênteses
    - **`return x+y;`** e não **`return (x+y);`**
  - O valor de retorno pode ser ignorado
    - por exemplo, a função **`printf`** retorna um valor, mas geralmente se ignora

Função

- Término de uma função
  - Ao encontrar a chave de fechamento
  - Ao ser retornada (**`return`**)

Exemplo

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

int imprimeRetornandoTamanho(char str[]) {
    printf("%s", str);
    return strlen(str); /* Nao precisa de parenteses, mas pode */
}

int main(int argc, char *argv[]) {
    char str[100];
    printf("Digite o nome: ");
    gets(str);
    imprimeRetornandoTamanho(str);
    return 0;
}
```

- Tamanho do *string* retornado pela função foi ignorado

Função

- Como já se sabe, arranjos podem ser passados como parâmetros para funções
- Contudo, ao se passar arranjos, além da função receber o arranjo, ela deve receber o **tamanho do arranjo**
  - Pois, dentro da função não funcionará corretamente a função **`sizeof`**
  - Exceção para arranjos de caracteres que estão representando *strings*, pois o tamanho do *string* é definido pelo caractere '\0' e é utilizada a função **`strlen`**

Exemplo

```
1  #include<stdio.h>

3  void imprimeArranjo(int v[], int tam) {
       int i;
5      for (i = 0; i < tam; i++) {
           printf("%4d", v[i]);
7      }
   }
9
   int main(int argc, char *argv[]) {
11     int v[4] = { 4, 9, 2, 3 };
       imprimeArranjo(v, sizeof(v) / sizeof(int));
13     return 0;
   }
```

- O tamanho deve ser passado

Exemplo

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

void imprimeString(char str[]) {
    int i, tam;
    tam = strlen(str);
    for (i = 0; i < tam; i++) {
        printf("%c", str[i]);
    }
}

int main(int argc, char *argv[]) {
    char v[10] = {'A', 'N', 'A', '\0', 'I', 'X', 'K', '^', '%', '!'};
    imprimeString(v);
    return 0;
}
```

- Não precisa passar o tamanho quando o arranjo de caracteres estiver representando um *string*

Exercícios de Fixação

- Implementar as funções:
  - `fatorial`
  - `escreveMaior`
  - `retornaMenorElemento`
  - `retornaMaiorElemento`
  - `retornaMedia`

Exercício - Fatorial

```
#include<stdio.h>

long int fat(int n) {
    int i;
    long int res = 1;
    if (n == 0 || n == 1) {
        return 1;
    }
    for (i = 2; i <= n; i++) {
        res *= i;
    }
    return res;
}

int main(int argc, char *argv[]) {
    int n;
    scanf("%d", &n);
    printf("%d! = %ld", n, fat(n));
    return 0;
}
```

Exercício - Escreve Maior

```
#include<stdio.h>

void escreveMaior(int n) {
    int i;
    for (i = n; i >= 1; i--) {
        printf("%d", i);
        printf((i > 1) ? ">" : "\n");
    }
}

int main(int argc, char *argv[]) {
    int n;
    scanf("%d", &n);
    escreveMaior(n);
    return 0;
}
```

Exercício - Escreve Maior (*hardcore*)

```
#include<stdio.h>

void escreveMaior(int n) {
    for (; n >= 1;printf("%d",n),printf(n-- > 1 ? " > " : "\n" ));
}

int main(int argc, char *argv[]) {
    int n;
    scanf("%d", &n);
    escreveMaior(n);
    return 0;
}
```

Exercício - Retorna Menor Elemento

```
#include<stdio.h>

int retornaMenorElemento(int v[], int tam) {
    int i;
    int menor = v[0];
    for (i = 1; i < tam; i++) {
        if (v[i] < menor) {
            menor = v[i];
        }
    }
    return menor;
}

int main(int argc, char *argv[]) {
    int v[5] = { 2, 4, 5, 1, 3 };
    int tam = sizeof(v) / sizeof(int);
    printf("Menor el.: %d", retornaMenorElemento(v, tam));
    return 0;
}
```

Ordem de Declaração

- Uma função pode ser implementada após seu ponto de chamada ou mesmo em um outro arquivo fonte
- Entretanto, para utilizá-la deve-se especificar, ao menos, o seu protótipo
- O protótipo de uma função consiste em especificar seu retorno, seu nome e o tipo dos seus parâmetros formais
  - Uma espécie de assinatura da função

Funciona?

```
1  #include<stdio.h>

3  int main(int argc, char *argv[]) {
       int a = 2, b = 3, total;
5      total = soma(a, b);
       printf("Soma: %d", total);
7      return 0;
   }
9
   int soma(int x, int y) {
11     return x + y;
   }
```

E agora?

```
1  #include<stdio.h>

3  /*Prototipo*/
   int soma(int, int);
5
   int main(int argc, char *argv[]) {
7      int a = 2, b = 3, total;
       total = soma(a, b);
9      printf("Soma: %d", total);
       return 0;
11 }

13 int soma(int x, int y) {
       return x + y;
15 }
```

- Com o uso de protótipo, a função **main** sabe que existe a função **soma**

Importante

É importante mencionar que o uso de protótipo não se limita apenas à função **main**, mas sim, à **qualquer** função que deseje acessar métodos declarados abaixo dela

```
1  #include<stdio.h>

3  /*Prototipo*/
   int ePar(int);
5  int resto(int, int);

7  int main(int argc, char *argv[]) {
       int n;
9      scanf("%d", &n);
       printf(((ePar(n)) ? "%d eh par" : "%d eh impar"), n);
11     return 0;
   }
13
   int ePar(int n) {
15     return resto(n, 2) ? 0 : 1; /* Acessando funcao declarada abaixo */
   }
17
   int resto(int x, int y) {
19     return x % y;
   }
```

Recursão

- Como se sabe, uma função pode chamar uma outra função
  - Essa "outra função" pode ser ela mesma
- Uma função que chama a si mesma é dita ser uma função recursiva
- Funções recursivas podem chamar a si mesmas direta ou indiretamente
  - `a( )` que chama `a( )`
  - `a( )` que chama `b( )` que chama `a( )`

Recursão

- Existem problemas que são naturalmente recursivos
- O mais clássico é o fatorial
  - `0! = 1! = 1`
  - **`n! = n * (n-1)!`**
- Ao se implementar uma função recursiva, o mais importante é definir o seu **ponto de parada**
- No exemplo do fatorial, a recursão para quando se chega no 0 ou 1

Exercícios de Fixação

- Implementar as funções:
  - `fatorial` **recursivo**
  - `fibonacci` **recursivo**
  - `escreveMaior` **recursivo**

Exercício - Fatorial Recursivo

```c
#include<stdio.h>

/*Prototipo*/
long int fat(int);

int main(int argc, char *argv[]) {
    int n;
    scanf("%d", &n);
    printf("%d! = %ld", n, fat(n));
    return 0;
}

long int fat(int n) {
    if (n == 0 || n == 1) {
        return 1; /* Ponto de Parada */
    }
    return n * fat(n - 1); /* Codigo da Recursao */
}
```

Exercício - Fibonacci Recursivo

```c
#include<stdio.h>

/*Prototipo*/
long int fib(int);

int main(int argc, char *argv[]) {
    int n;
    scanf("%d", &n);
    printf("fib(%d) = %ld", n, fib(n));
    return 0;
}

long int fib(int n) {
    if (n == 0 || n == 1) { /* Ponto de Parada */
        return n;
    }
    return fib(n - 1) + fib(n - 2); /* Codigo da Recursao */
}
```

Algumas funções da biblioteca padrão

- **Funções de E/S** (`stdio.h`)
  - `printf()`, `scanf()`, `fprintf()`, `fscanf()`, `gets()`
- **Funções Matemáticas** (`math.h`)
  - `sin()`, `cos()`, `exp()`, `log()`, `pow()`, `sqrt()`
- **Teste e manipulação de caracteres** (`ctype.h`)
  - `isDigit()`, `isAlpha()`, `toupper()`

Algumas funções da biblioteca padrão

- **Funções de propóstio geral** (`stdlib.h`)
  - `malloc()`, `free()`, `exit()`, `rand()`
- **Manipulação de *strings* e arranjos** (`string.h`)
  - `strcpy()`, `strcmp()`, `strcat()`, `memcpy()`
- **Manipulação de datas e horas** (`time.h`)
  - `localtime()`, `time()`, `clock()`

Sintaxe Básica

Estruturas

Estruturas (Struct)

- O que conhecemos como **`record`** em Pascal, em C é conhecido como **`struct`**
- Uma estrutura é uma coleção de variáveis referenciadas por um nome, fornecendo uma maneira conveniente de atribuir informações (variáveis) relacionadas de forma agrupada
- A definição de uma estrutura é um modelo a ser seguido por todas as variáveis de seu tipo
- As *variáveis* que compreendem a estrutura são também conhecidas como *campos* ou *atributos* da estrutura

O exemplo abaixo cria uma estrutura para representação de uma data:

```
struct data {
    int dia;
    int mes;
    int ano;
}
```

Para declarar uma variável do tipo `data`, faz-se como abaixo:

- `struct data d;`

Instrução `typedef`

- A instrução **`typedef`** permite denotar novos nomes à linguagem
- Assim, pode-se utilizar o comando **`typedef`** para que se simplificar a declaração de variáveis de estrutura

Exemplo

```
1 typedef struct {
      int dia;
3     int mes;
      int ano;
5 } data;
```

Assim, a estrutura é simplesmente conhecida como `data` e declaração se faz como um tipo primitivo (`char`, `int` etc)

- **`data`** d;

Exemplo `struct` – Atribuição e Acesso aos elementos

```
1  #include<stdio.h>

3  typedef struct {
       int dia;
5      int mes;
       int ano;
7  } data;

9  int main(int argc, char *argv[]) {
       data d;
11     d.dia = 31;
       d.mes = 12;
13     d.ano = 1999;

15     printf("A data e %2.d/%2.d/%4.d\n", d.dia, d.mes, d.ano);

17     return 0;
   }
```

Exemplo `struct` – Leitura e Acesso aos elementos

```
#include<stdio.h>

typedef struct {
    int dia;
    int mes;
    int ano;
} data;

int main(int argc, char *argv[]) {
    data d;

    scanf("%d", &d.dia);
    scanf("%d", &d.mes);
    scanf("%d", &d.ano);

    printf("A data e %2.d/%2.d/%4.d\n", d.dia, d.mes, d.ano);

    return 0;
}
```

Exemplos (`struct`)

- Vamos a seguir, ver e entender mais exemplos utilizando estruturas

Exemplo 1

- Um exemplo que possui uma estrutura chamada **`cliente`** e que, dentro dessa estrutura, possui um campo do tipo **`endereco`** (outra estrutura), i.e., um cliente possui suas informações e ainda possui um endereço

- Nesse exemplo, serão lidos todos os dados de um cliente e depois serão impressos os dados lidos (observe a notação ponto para acessar os membros de uma estrutura)

Sintaxe Básica – Estruturas

```c
#include<stdio.h>

typedef struct {
    char logradouro[60];
    int numero;
} endereco;

typedef struct {
    int codigo;
    char nome[60];
    endereco e;
} cliente;

int main(int argc, char *argv[]) {
    cliente c;

    printf("Codigo: ");
    scanf("%d", &c.codigo);
    printf("Nome: ");
    gets(c.nome);
    printf("Logradouro: ");
    gets(c.e.logradouro);
    printf("Numero: ");
    scanf("%d", &c.e.numero);

    printf("Cliente %d: %s reside em %s, %d\n", c.codigo, c.nome,
            c.e.logradouro, c.e.numero);

    return 0;
}
```

Exemplo 2

- Um exemplo que possui uma estrutura chamada `carro` e que, dentro dessa estrutura, possui um campo do tipo `proprietario` (outra estrutura), i.e., um carro possui suas informações e ainda possui um proprietário

- Nesse exemplo, serão atribuídos todos os dados de um carro e depois serão impressos os dados atribuídos (observe a notação ponto para acessar os membros de uma estrutura)

Sintaxe Básica – Estruturas

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

typedef struct {
    int rg;
    char nome[60];
} proprietario;

typedef struct {
    char modelo[30];
    int ano;
    char placa[7];
    proprietario p;
} carro;

int main(int argc, char *argv[]) {
    carro c;

    strcpy(c.modelo,"GOL");
    c.ano = 2010;
    strncpy(c.placa,"BAD0007",7);
    c.p.rg = 10200300;
    strcpy(c.p.nome,"Ricardo Terra");

    printf("CARRO: %s, ano %d, placa %.7s\n", c.modelo, c.ano, c.placa);
    printf("PROPRIETARIO: %s (RG: %d)", c.p.nome, c.p.rg);

    return 0;
}
```

Dica:

- A atribuição pode ser simplificada
  - Observe declaração da variável **carro**

```
1  #include<stdio.h>
   #include<string.h>
3
   typedef struct {
5      int rg;
       char nome[60];
7  } proprietario;

9  typedef struct {
       char modelo[30];
11     int ano;
       char placa[7];
13     proprietario p;
   } carro;
15
   int main(int argc, char *argv[]) {
17     carro c = { "GOL", 2010, "BAD0007", { 10200300, "Ricardo Terra" } };

19     printf("CARRO: %s, ano %d, placa %.7s\n", c.modelo, c.ano, c.placa);
       printf("PROPRIETARIO: %s (RG: %d)", c.p.nome, c.p.rg);
21
       return 0;
23 }
```

Exemplo 2

- Um exemplo que possui uma estrutura chamada **`aluno`** e que, dentro dessa estrutura, possui um campo do tipo **`disciplina`** (outra estrutura) que possui um campo do tipo **`professor`** (outra estrutura), i.e., um aluno possui uma única disciplina que está vinculada a um professor

- Nesse exemplo, serão atribuídos todos os dados de um aluno e depois serão impressos os dados lidos (observe a notação ponto para acessar os membros de uma estrutura)

Sintaxe Básica – Estruturas

```c
#include<stdio.h>

typedef struct {
    int ctps;
    char nome[60];
} professor;

typedef struct {
    int codigo;
    char nome[60];
    professor prof;
} disciplina;

typedef struct {
    int matricula;
    char nome[60];
    disciplina disc;
} aluno;

int main(int argc, char *argv[]) {
    aluno a = { 40, "JORGE", { 1, "ED-I", { 1001, "Terra" } } };

    printf("%d\n", a.matricula);
    printf("%s\n", a.nome);
    printf("%d\n", a.disc.codigo);
    printf("%s\n", a.disc.nome);
    printf("%d\n", a.disc.prof.ctps);
    printf("%s\n", a.disc.prof.nome);

    return 0;
}
```

Exemplo 3

- Um exemplo que manipula um arranjo de uma estrutura chamada `fitness` que possui apenas `peso` e `altura`

- Nesse exemplo, serão lidos os dados *fitness* de 10 pessoas e, para cada pessoa, será invocada uma função que retorna o seu Índice de Massa Corporal (IMC)

Sintaxe Básica – Estruturas

```c
#include<stdio.h>
#include<math.h>

typedef struct {
    float peso; /* em kg */
    float altura; /* em m */
} fitness;

float imc(fitness); /* Prototipo */

int main(int argc, char *argv[]) {
    fitness v[10];
    int i, tam = sizeof(v) / sizeof(fitness);

    for (i = 0; i < tam; i++) {
        scanf("%f", &v[i].peso);
        scanf("%f", &v[i].altura);
    }

    for (i = 0; i < tam; i++) {
        printf("IMC pessoa %d: %f\n", i + 1, imc(v[i]));
    }

    return 0;
}

float imc(fitness f) {
    return f.peso / pow(f.altura, 2);
}
```

3. Ponteiros – Conteúdo

Ponteiros

Conceitualização

O que são ponteiros?

- Um ponteiro é uma variável que contém um endereço de memória
  - Esse endereço é normalmente a posição de uma outra variável na memória
- Se uma variável contém o endereço de uma outra, então a primeira variável é dita apontar para a segunda
  - Eis a origem do nome ponteiro

Variáveis ponteiros?

- Se uma variável é de fato um ponteiro, ela deve ser declarada como tal. A forma geral para declarar uma variável ponteiro é:
  - `tipo *nome;`
- Um ponteiro aponta para uma outra variavél. Contudo, em sua inicialização – como em todos os tipos de variáveis – seu valor inicial é lixo de memória. Utilizando `NULL`, pode-se criar um ponteiro e apontá-lo para nulo
  - `int *p = NULL;`
- Para imprimir o endereço de memória, utiliza-se **%p**
  - `printf("O ponteiro p aponta para %p", p);`

Os operadores de ponteiros

- Inicialmente, existem dois operadores especiais para ponteiros: `*` e `&`
- O operador `&` é um operador unário que devolve o endereço na memória do seu operando

Exemplo

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    int count = 100;
    int *p;

    p = &count;

    printf("O endereco apontado por p e %p", p);

    return 0;
}
```

Os operadores de ponteiros

- É colocado em **p** o endereço da memória que contém a variável **count**
  - Esse endereço é a posição interna ao computador daquela variável
- O endereço não tem relação alguma com o valor de **count**
- O operador **&** pode ser imaginado como "*o endereço de*", assim como o comando de atribuição do último exemplo significa "***p*** *recebe o endereço de* ***count***"

Os operadores de ponteiros

- O operador * é o complemento de &. É um operador unário que devolve o valor da variável localizada no endereço que o segue
  - `int q = *p;`
- Sabendo que **p** contém o endereço da variável **count**, i.e., **p** aponta para **count** podemos pensar no resultado de duas maneiras:
  - o exemplo coloca o valor de **count** em **q**
  - o exemplo coloca o valor da variável apontada por **p** em **q**
- O operador * pode ser imaginado como "*o valor da variável apontada por*", assim como o exemplo acima significa "**q** recebe o valor da variável apontada por **p**"

Exemplo

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    int count = 100;
    int *p;

    p = &count;

    printf("%d", *p);     /* Imprimira 100 */
    *p = 34;              /* Altera valor de count para 34 */
    printf("%d", count); /* Imprimira 34 */
    printf("%d", *p);     /* Imprimira 34 */

    return 0;
}
```

- Observe que pode-se imprimir ou alterar o valor a partir do ponteiro para a variável

Exemplo 1

```
1 #include<stdio.h>

3 int main(int argc, char *argv[]) {
      int x = 2;
5     int *p;

7     p = &x; /* Ponteiro recebe "o endereco de" x */
      *p = 4; /* "O valor da variavel apontada por p" recebe 4 */
9
      printf("valor de x: %d\n", x); /* Imprimira 4 */
11
      return 0;
13 }
```

Exemplo 2

```
1  #include<stdio.h>

3  int main(int argc, char *argv[]) {
       int x = 2;
5      int *p1, *p2;

7      /* Um ponteiro so recebe um endereco ou um outro ponteiro */
       p1 = &x;
9      p2 = p1;

11     printf("%p = %p? Apontam para o mesmo endereco?\n", p1, p2);

13     printf("Valor apontado por p1: %d\n", *p1); /* Imprimira 2 */
       printf("Valor apontado por p2: %d\n", *p2); /* Imprimira 2 */
15
       return 0;
17 }
```

Passagem por valor

- A convenção de C de passagem de parâmetros é por valor, isto é, quando se passa o valor de uma variável para uma função, tem-se a certeza que seu valor não será alterado

Exemplo

```
1  #include<stdio.h>

3  void incrementa(int x) {
       x++; /* Incrementa x */
5  }

7  int main(int argc, char *argv[]) {
       int x = 10;
9      incrementa(x); /* Passei o valor de x */
       printf("valor de x: %d\n", x); /* Imprimira 10 */
11     return 0;
   }
```

Simulando uma chamada por referência

- Muito embora a convenção de C de passagem de parâmetros seja por valor, você pode **simular** uma chamada por referência
- Para isso, passa-se o endereço de uma variável para um ponteiro (parâmetro formal de uma função)
- Isso faz com que o endereço da variável seja passado à função e, desse modo, você pode alterar o valor da variável fora da função
  - Contudo, isso não se qualifica como passagem por referência, uma vez que o endereço da variável é **copiado** para o ponteiro, i.e., ainda é passagem por valor

Simulando uma chamada por referência

- Endereços de variáveis são passados para as funções de forma simples
  - Obviamente é necessário declarar os parâmetros formais da função como do tipo ponteiro

Exemplo

```
#include<stdio.h>

void incrementa(int *x) {
    (*x)++; /* Incrementa o valor da variavel apontada por x */
}

int main(int argc, char *argv[]) {
    int x = 10;
    incrementa(&x); /* Passei o endereco de x */
    printf("valor de x: %d\n", x); /* Imprimira 11 */
    return 0;
}
```

Exemplo - Função de troca de valores

```
#include<stdio.h>

void troca(int *, int *); /* Prototipo */

int main(int argc, char *argv[]) {
    int x = 2, y = 4;

    printf("x = %d e y = %d\n", x, y);
    troca(&x, &y);
    printf("x = %d e y = %d\n", x, y);

    return 0;
}

void troca(int *px, int *py) { /* Observe o uso do asterisco (*) */
    int temp = *px;
    *px = *py;
    *py = temp;
}
```

Exemplo - Função de troca de valores (*hardcore*)

```
1  #include<stdio.h>

3  void troca(int *, int *); /* Prototipo */

5  int main(int argc, char *argv[]) {
       int x = 2, y = 4;
7
       printf("x = %d e y = %d\n", x, y);
9      troca(&x, &y);
       printf("x = %d e y = %d\n", x, y);
11
       return 0;
13 }

15 void troca(int *px, int *py) { /* sem uso de variavel auxiliar */
       *px += *py; /* x = (x + y) */
17     *py = *px - *py; /* y = (x + y) - y = x */
       *px -= *py; /* x = (x + y) - x = y */
19 }
```

Ponteiros

Aritmética

Arranjos e Ponteiros

- Arranjos e ponteiros são intimamente relacionados em C
- O **nome** de um **arranjo** é um ponteiro **constante** para o primeiro elemento do arranjo

Exemplo:

- `double v[5];`
- Logo, as seguintes sintaxes são equivalentes:
  - `*v` $\leftrightarrow$ `v[0]`
  - `v` $\leftrightarrow$ `&v[0]`

Logo, são assinaturas equivalentes de funções:

- `void funcao (float v[], int tam)`

  ou

- `void funcao (float *v, int tam)`

Logo, são chamadas equivalentes às funções:

- `funcao(v, tam);`

  ou

- `funcao(&v[0], tam);`

Possíveis operações:

Sejam `p`, $\texttt{p}_a$ e $\texttt{p}_b$ ponteiros e `i` um número inteiro

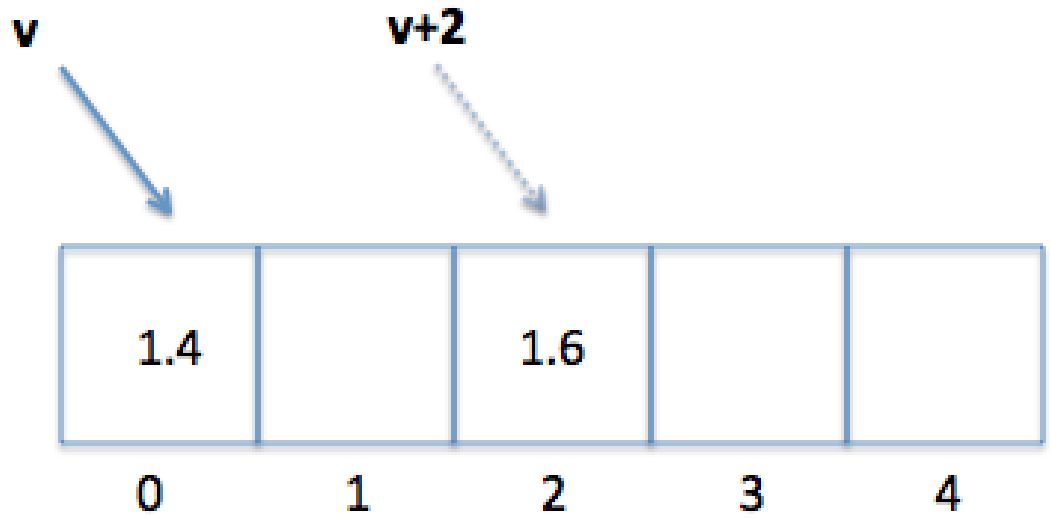
- `p + i` $\rightarrow$ ponteiro `p` é deslocado `i` elementos para direita
- `p - i` $\rightarrow$ ponteiro `p` é deslocado `i` elementos para esquerda
- $\texttt{p}_a$ - $\texttt{p}_b$ $\rightarrow$ retorna a distância, em elementos, entre os ponteiros $\texttt{p}_a$ e $\texttt{p}_b$

Portanto, segue-se que:

- `double v[5];`

  `v[0] = 1.4;` ↔ `*v = 1.4;`

  `v[2] = 1.6;` ↔ `*(v+2) = 1.6;`



Aritmética de Ponteiros: + e -

- Movimentam um ponteiro levando em consideração o tamanho do tipo do elemento

Observe:

```
1 float v[5];
  float *p;
3 p = v;        /*Ponteiro p está apontando para v[0]*/
  p = v + 2;    /*Ponteiro p está apontando para v[2]*/
5 p--;          /*Ponteiro p está apontando para v[1]*/
```

Exemplo 1

- Assim que se **iterava** um arranjo:

```
1 #include<stdio.h>

3 int main(int argc, char *argv[]) {
      int i, v[] = { 13, 15, 17, 19, 21 };
5     int tam = sizeof(v) / sizeof(int);

7     for (i = 0; i < tam; i++) {
          printf("%d\t", v[i]);
9     }

11    return 0;
  }
```

Exemplo 2 - Arranjo de inteiros

- Agora, assim que se **itera** um arranjo:

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    int i, v[] = { 13, 15, 17, 19, 21 };
    int tam = sizeof(v) / sizeof(int);
    int *p;

    for (i = 0, p = v; i < tam; i++, p++) {
        printf("%d\t", *p);
    }

    return 0;
}
```

Exemplo 2 - Arranjo de caracteres

- Idêntico ao anterior, porém o arranjo é de caracteres

```
1  #include<stdio.h>

3  int main(int argc, char *argv[]) {
       char vogais[] = { 65, 'E', 'I', 79, 'U' };
5      int i, tam = sizeof(vogais) / sizeof(char);
       char *p;
7
       for (i = 0, p = vogais; i < tam; i++, p++) {
9          printf("%c\t", *p);
       }
11
       return 0;
13 }
```

Exemplo 3

- Se além de um ponteiro para o primeiro elemento tivermos um outro ponteiro para o último elemento, pode-se iterar sem as variáveis **i** e **tam**:

```
1  #include<stdio.h>

3  int main(int argc, char *argv[]) {
       int v[] = { 13, 15, 17, 19, 21 };
5      int *p, *q;
       int tam = sizeof(v) / sizeof(int);
7
       for (p = v, q = v + tam - 1; p <= q; p++) {
9          printf("%d\t", *p);
       }
11
       return 0;
13 }
```

Ponteiros para estrutura

- A linguagem C ANSI permite ponteiros para estruturas **exatamente** como permite ponteiros para os outros tipos de variáveis, como **char**, **int**, **float**...
- Sintaxe:
  - `<nome-da-estrutura> *p;`
- Exemplos:
  - `pessoa *p;`
  - `aluno *p;`
  - `disciplina *p;`

Ponteiros para estrutura

- Para acessar os membros de uma estrutura utilizando um ponteiro para aquela estrutura, você **pode** e **deve** usar o operador **->**
- Por exemplo, seja **p** um ponteiro para **conta**. Assim:
  - `p->saldo` equivale a `(*p).saldo`
- O operador seta (**->**) substitui o asterisco e o ponto
- É importante mencionar que só funciona quando se tem um **ponteiro para estrutura**

Exemplo 4 - Manipulando estruturas através de um ponteiro

```
1  #include<stdio.h>

3  typedef struct {
       int numero;
5      char correntista[60];
       float saldo;
7  } conta;

9  int main(int argc, char *argv[]) {
       conta c;
11     conta *p;
       p = &c;
13
       scanf("%d", &(p->numero));
15     gets(p->correntista);
       scanf("%f", &(p->saldo));
17
       printf("(%d) %s tem $%.2f", p->numero, p->correntista, p->saldo);
19
       return 0;
21 }
```

Exemplo 5 - Manipulando estruturas através de um ponteiro

```c
#include<stdio.h>

typedef struct {
    int numero;
    char correntista[60];
    float saldo;
} conta;

/* Prototipo */
void transferir(conta *origem,
                conta *destino,
                float valor);

int main(int argc, char *argv[]) {
    conta contaA = { 1, "CARLOS MACIEL", 300.00 };
    conta contaB = { 2, "RICARDO TERRA", 200.00 };

    transferir(&contaA, &contaB, 40.00);

    printf("(%d) %s agora tem $%.2f\n",
           contaA.numero, contaA.correntista,
           contaA.saldo);
    printf("(%d) %s agora tem $%.2f\n",
           contaB.numero, contaB.correntista,
           contaB.saldo);

    return 0;
}

void transferir(conta *origem,
                conta *destino,
                float valor) {
    origem->saldo -= valor;
    destino->saldo += valor;
}
```

Exemplo 6 - Manipulando estruturas através de um ponteiro

```
1  #include<stdio.h>

3  typedef struct {
       int matricula;
5      char nome[60];
   } aluno;
7
   /*Prototipos*/
9  void le(aluno *p);
   void imprime(aluno *p);
11
   int main(int argc, char *argv[]) {
13     aluno a;
       le(&a);
15     imprime(&a);
       return 0;
17 }

19 void le(aluno *p) {
       scanf("%d", &(p->matricula));
21     gets(p->nome);
   }
23
   void imprime(aluno *p) {
25     printf("%d %s", p->matricula, p->nome);
   }
```

Exemplo 7 - Manipulando estruturas que possuem outras estruturas

```c
#include<stdio.h>

typedef struct {
    int codigo;
    char nome[60];
} disciplina;

typedef struct {
    int matricula;
    char nome[60];
    disciplina d;
} aluno;

/*Prototipos*/
void le(aluno *p);
void imprime(aluno *p);
```

```c
int main(int argc, char *argv[]) {
    aluno a;

    le(&a);
    imprime(&a);

    return 0;
}

void le(aluno *p) {
    scanf("%d", &(p->matricula));
    gets(p->nome);
    scanf("%d", &(p->d.codigo));
    gets(p->d.nome);

}

void imprime(aluno *p) {
    printf("%d %s %d %s",
           p->matricula,p->nome,p->d.codigo,p->d.nome);

}
```

Exemplo 8 - Manipulando arranjos de estruturas

```c
#include<stdio.h>

typedef struct {
    int matricula;
    char nome[60];
} aluno;

/*Prototipos*/
void le(aluno *v, int tam);
void imprime(aluno *v, int tam);

int main(int argc, char *argv[]) {
    aluno v[3];
    int tam = sizeof(v)/
              sizeof(aluno);

    le(v, tam);
    imprime(v, tam);

    return 0;
}
```

```c
void le(aluno *v, int tam) {
    int i;
    aluno *p;
    for (i = 0, p = v; i < tam; i++, p++) {
        scanf("%d", &(p->matricula));
        gets(p->nome);
    }
}

void imprime(aluno *v, int tam) {
    int i;
    aluno *p;
    for (i = 0, p = v; i < tam; i++, p++) {
        printf("[%d] %d %s\n",
               i, p->matricula, p->nome);
    }
}
```

Ponteiros

Alocação Dinâmica de Memória

Alocação Dinâmica de Memória

- Estruturas de dados estáticas têm seu espaço pré-determinado a tempo de compilação
- Então, como criar um arranjo que saiba o tamanho durante a execução do programa?

Por exemplo, isto está ERRADO!

```
1 int main(int argc, char *argv[]) {
      int x;
3     scanf("%d", &x);

5     int v[x]; /*Completamente Errado*/

7     /* ... */

9     return 0;
  }
```

Alocação Dinâmica de Memória

- Assim, estruturas dinâmicas têm seu tamanho determinado em tempo de execução
- Para isso, existem mecanismos que permitam fazer alocação dinâmica de memória em tempo de execução
- Memória pode ser alocada de uma região chamada *heap*
- Memória alocada da *heap* **DEVE** ser devolvida quando não for mais utilizada

Funções para manipulação da *heap*

- Encontram-se na biblioteca **`stdlib.h`**
  - **`malloc, calloc, realloc, free...`**

Função `malloc`

- Solicita memória ao sistema

Sintaxe:

- `void* malloc (int tam)`
  - **`void*`** → ponteiro de tipo não especificado – necessita de *typecast*
  - **`tam`** → quantidade de *bytes* a serem alocados

- Retorno de **`malloc`** é um ponteiro sem tipo para o início da área alocada

Função `free`

- Libera memória ao sistema

Sintaxe:

- `void free (void *p)`
  - **`void *p`** → um ponteiro que aponta para uma área alocada com **`malloc`**, ou melhor, deve ser passado o endereço do início da área alocada pelo **`malloc`**

Exemplo 1: Alocação de 100 caracteres (Observe o *typecast*)

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

int main(int argc, char *argv[]) {
    char *pc;
    pc = (char*) malloc(100 * sizeof(char));

    gets(pc);
    printf("%s", pc);

    free(pc);
    return 0;
}
```

Pergunta:

- Sempre conseguirá alocar a memória desejada?

IMPORTANTE:

- Em caso de erro, isto é, não conseguir memória disponível para efetuar a alocação, a função **`malloc`** retorna **`NULL`**

Exemplo 2: Alocação de `x` inteiros (Observe o *typecast*)

```
1  #include<stdio.h>
   #include<stdlib.h>
3
   int main(int argc, char *argv[]) {
5      int x;
       int *pi;
7      scanf("%d", &x);
       pi = (int*) malloc(x * sizeof(int));
9
       if (pi == NULL) {
11         printf("Memoria Insuficiente. Programa encerrado.");
           return -1;
13     }

15     pi[5] = 44; /*Considere x > 5*/

17     /*varias outras operacoes*/

19     free(pi);
       return 0;
21 }
```

Exemplo 3: Alocação de x pontos flutuantes de precisão dupla

(Observe o teste)

```
1  #include<stdio.h>
   #include<stdlib.h>
3
   int main(int argc, char *argv[]) {
5      int x;
       double *pd;
7      scanf("%d", &x);
       pd = (double*) malloc(x * sizeof(double));
9
       if (!pd) { /*Mais Profissional*/
11         printf("Memoria Insuficiente. Programa encerrado.");
           return -1;
13     }

15     pd[5] = 44; /*Considere x > 5*/

17     /*varias outras operacoes*/

19     free(pd);
       return 0;
21 }
```

Exemplo Completo: Alocação, inicialização e liberação de um arranjo de ponto flutuante

```
1  #include<stdio.h>
   #include<stdlib.h>
3
   int main(int argc, char *argv[]) {
5      float *v;
       int tam, i;
7      scanf("%d", &tam);

9      v = (float*) malloc(tam * sizeof(float));
       if (!v) { /*Verifica se conseguiu alocar*/
11         printf("Memoria Insuficiente. Programa encerrado.");
           return -1;
13     }

15     for (i = 0; i < tam; i++) { /*Inicializa o arranjo*/
           v[i] = 0;
17     }

19     /*varias outras operacoes*/

21     free(v); /*Libera a area de memoria alocada*/
       return 0;
23 }
```

Exemplo Anterior já com Aritmética de Ponteiros

```
1  #include<stdio.h>
   #include<stdlib.h>
3
   int main(int argc, char *argv[]) {
5      float *v, *p;
       int tam, i;
7      scanf("%d", &tam);

9      v = (float*) malloc(tam * sizeof(float));
       if (!v) { /*Verifica se conseguiu alocar*/
11         printf("Memoria Insuficiente. Programa encerrado.");
           return -1;
13     }

15     for (p = v, i = 0; i < tam; i++, p++) { /*Inicializa o arranjo*/
           *p = 0;
17     }

19     /*varias outras operacoes*/

21     free(v); /*Libera a area de memoria alocada*/
       return 0;
23 }
```

Exemplo Completo: Alocação, inicialização e liberação de um arranjo de estrutura com Aritmética de Ponteiros

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

typedef struct {
    int matricula;
    char nome[60];
} aluno;

int main(int argc, char *argv[]) {
    aluno *v, *p;
    int tam, i;
    scanf("%d", &tam);

    v = (aluno*)
        malloc(tam * sizeof(aluno));

    /*Verifica se conseguiu alocar*/
    if (!v) {
        printf("Mem. Insuficiente.");
        return -1;
    }
```

```c
    /*Inicializa o arranjo*/
    for (p = v, i = 0; i < tam; i++, p++) {
        scanf("%d", &(p->matricula));
        gets(p->nome);
    }

    /*Imprime o arranjo*/
    for (p = v, i = 0; i < tam; i++, p++) {
        printf("%d - %s\n", p->matricula, p->nome);
    }

    /*Libera a area de memoria alocada*/
    free(v);

    return 0;
}
```

Ponteiros

Alocação Dinâmica de Arranjos Multidimensionais

Ponteiro para ponteiro

- Como sabemos, um arranjo de inteiros é um ponteiro para inteiros:
  - `int vA[]` ↔ `int *vA`
- Logo, um arranjo bidimensional de inteiros é um arranjo de ponteiros para inteiros:
  - `int vA[][]` ↔ `int *vA[]` ↔ `int **vA`
- É por isso que a assinatura do método **`main`** pode ser:
  - `int main (int argc, char **argv[])`
  - `int main (int argc, char *argv[] )`
  - `int main (int argc, char **argv )`

Exemplo 1

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    float f1 = 27, f2 = 13, *pf1, *pf2, **ppf1, **ppf2;

    pf1 = &f1;
    pf2 = &f2;
    printf("%.2f %.2f\n", *pf1, *pf2); /*Imprimira 27.00 13.00*/

    ppf1 = &pf1;
    ppf2 = &pf2;

    **ppf1 = *pf1 - 1;
    **ppf2 = *pf2 + 1;

    printf("%.2f %.2f\n", **ppf1, **ppf2); /*Imprimira 26.00 14.00*/

    return 0;
}
```

Alocação Dinâmica de Arranjos Multidimensionais

- Assim como é possível com um arranjo, pode-se alocar arranjos multidimensionais dinamicamente

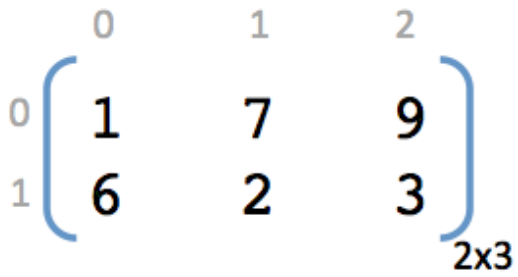
- Então, como criar um arranjo mutidimensional que se saiba seu tamanho durante a execução do programa?

Por exemplo, isto está ERRADO!

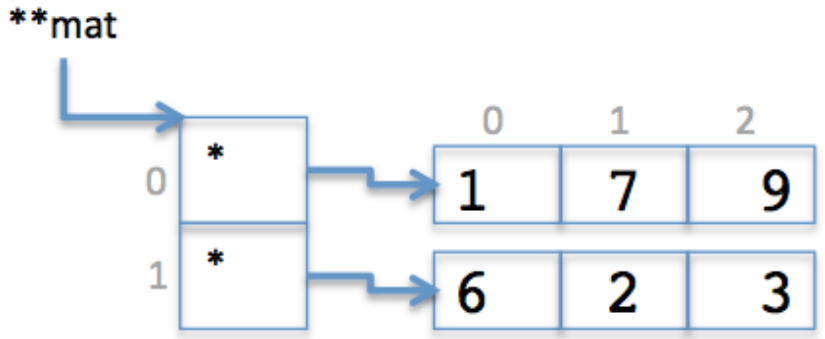
```
1 int main(int argc, char *argv[]) {
      int x, y;
3     scanf("%d %d", &x, &y);

5     int mat[x][y]; /*Completamente Errado*/

7     /* ... */

9     return 0;
  }
```

Matriz

- Matriz é um arranjo bidimensional. Na verdade, pode ser pensado como um arranjo de arranjos
- A imagem abaixo apresenta como é visualmente raciocinada uma matriz e como é a sua respectiva estrutura de dados



(a) Visual



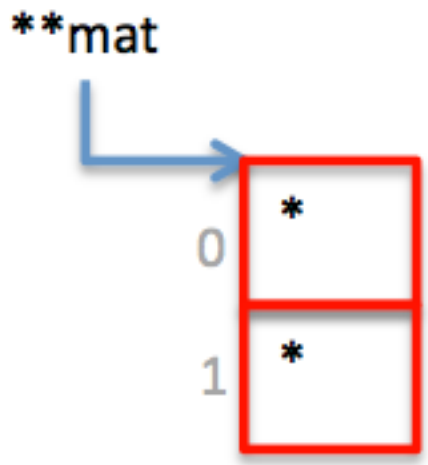
(b) Estrutura de Dados

Exemplo 2: Alocação de matriz x x y

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

/*Prototipo*/
int soma(int **mat, int x, int y);

int main(int argc, char *argv[]) {
    int **mat;
    int x, y, i, j;

    scanf("%d", &x);
    scanf("%d", &y);

    mat = (int**) malloc(x * sizeof(int*));
    if (!mat) {
        printf("MEM. INSUFICIENTE.");
        return -1;
    }

    for (i = 0; i < x; i++) {
        mat[i] = (int*) malloc(y *
                               sizeof(int));
        if (!mat[i]) {
            printf("MEM. INSUFICIENTE.");
            return -1;
        }
    }

    for (i = 0; i < x; i++) {
        for (j = 0; j < y; j++) {
            scanf("%d", &mat[i][j]);
        }
    }

    printf("SOMA DOS ELEMENTOS: %d",
                            soma(mat,x,y));

    for (i = 0; i < x; i++) {
        free(mat[i]);
    }

    free(mat);

    return 0;
}

int soma(int **mat, int x, int y) {
    int i, j, soma = 0;
    for (i = 0; i < x; i++) {
        for (j = 0; j < y; j++) {
            soma += mat[i][j];
        }
    }
    return soma;
}
```

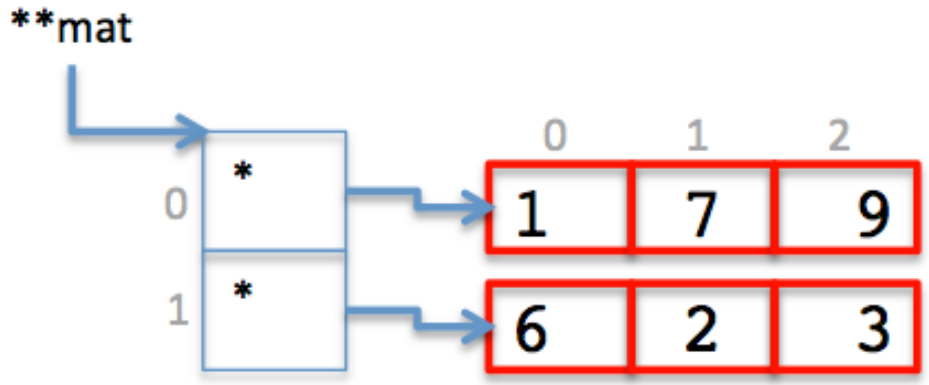
Observe o que faz o seguinte código (suponha x = 2):

```
1       mat = (int**) malloc(x * sizeof(int*));
```



E o que faz o seguinte código de repetição (suponha $y = 3$):

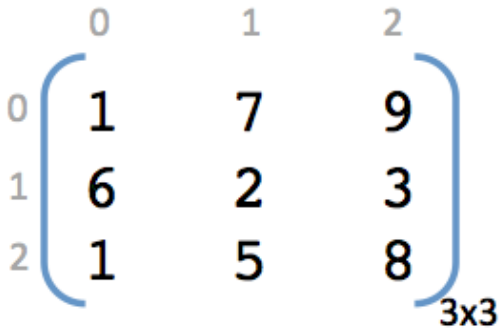
```
    for (i = 0; i < x; i++) {
2       mat[i] = (int*) malloc(y *
                                sizeof(int));
4       if (!mat[i]) {
            printf("MEM. INSUFICIENTE.");
6           return -1;
        }
8   }
```



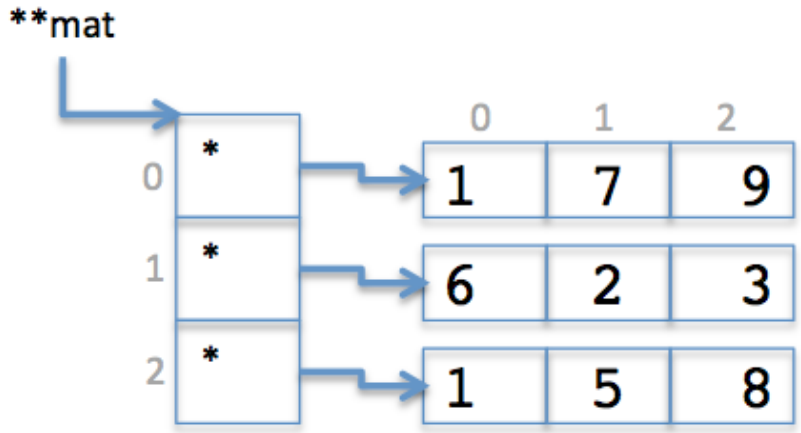
Matriz quadrada

- Matriz quadrada é um tipo especial de matriz cujo número de linhas é igual ao número de colulas

- A imagem abaixo apresenta como é visualmente raciocinada uma matriz quadrada e como é a sua respectiva estrutura de dados



(a) Visual



(b) Estrutura de Dados

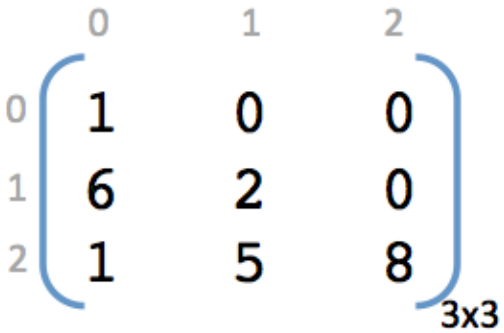
Exemplo 3: Alocação de matriz quadrada

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

/*Prototipo*/
int soma(int **mat, int tam);

int main(int argc, char *argv[]) {
    int **mat;
    int tam, i, j;

    scanf("%d", &tam);

    mat = (int**) malloc(tam * sizeof(int*));
    if (!mat) {
        printf("MEM. INSUFICIENTE.");
        return -1;
    }

    for (i = 0; i < tam; i++) {
        mat[i] = (int*) malloc(tam *
                               sizeof(int));
        if (!mat[i]) {
            printf("MEM. INSUFICIENTE.");
            return -1;
        }
    }
    for (i = 0; i < tam; i++) {
        for (j = 0; j < tam; j++) {
            scanf("%d", &mat[i][j]);
        }
    }

    printf("SOMA DOS ELEMENTOS: %d",
                                soma(mat,tam));

    for (i = 0; i < tam; i++) {
        free(mat[i]);
    }

    free(mat);

    return 0;
}

int soma(int **mat, int tam) {
    int i, j, soma = 0;
    for (i = 0; i < tam; i++) {
        for (j = 0; j < tam; j++) {
            soma += mat[i][j];
        }
    }
    return soma;
}
```

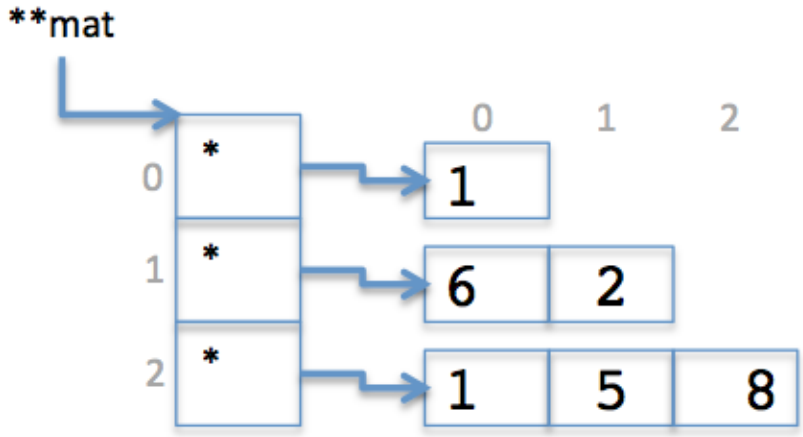
Matriz triangular

- Matriz triangular é um tipo especial de matriz quadrada em que todos os elementos acima (triangular superior) ou abaixo (triangular inferior) da diagonal principal são nulos

- A imagem abaixo apresenta como é visualmente raciocinada uma matriz triangular inferior e como é a sua respectiva estrutura de dados



(a) Visual



(b) Estrutura de Dados

Exemplo 4: Alocação de matriz triangular inferior

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

/*Prototipo*/
int soma(int **mat, int tam);

int main(int argc, char *argv[]) {
    int **mat;
    int tam, i, j;

    scanf("%d", &tam);

    mat = (int**) malloc(tam * sizeof(int*));
    if (!mat) {
        printf("MEM. INSUFICIENTE.");
        return -1;
    }

    for (i = 0; i < tam; i++) {
        mat[i] = (int*) malloc((i+1) *
                               sizeof(int));
        if (!mat[i]) {
            printf("MEM. INSUFICIENTE.");
            return -1;
        }
    }
    for (i = 0; i < tam; i++) {
        for (j = 0; j < (i + 1); j++) {
            scanf("%d", &mat[i][j]);
        }
    }

    printf("SOMA DOS ELEMENTOS: %d",
           soma(mat,tam));

    for (i = 0; i < tam; i++) {
        free(mat[i]);
    }

    free(mat);

    return 0;
}

int soma(int **mat, int tam) {
    int i, j, soma = 0;
    for (i = 0; i < tam; i++) {
        for (j = 0; j < (i + 1); j++) {
            soma += mat[i][j];
        }
    }
    return soma;
}
```

Exemplo 3 *sem indexação*: Alocação de matriz quadrada

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

/*Prototipo*/
int soma(int **mat, int tam);

int main(int argc, char *argv[]) {
    int **mat, **pp, *p;
    int tam, i, j;

    scanf("%d", &tam);

    mat = (int**) malloc(tam * sizeof(int*));
    if (!mat) {
        printf("MEM. INSUFICIENTE.");
        return -1;
    }

    for (pp = mat,i = 0; i < tam; i++, pp++) {
        *pp = (int*) malloc(tam *
                            sizeof(int));
        if (!(*pp)) {
            printf("MEM. INSUFICIENTE.");
            return -1;
        }
    }
    for (pp=mat, i=0; i<tam; i++, pp++){
        for (p=*pp,j=0; j<tam; j++, p++){
            scanf("%d",p);
        }
    }

    printf("SOMA DOS ELEMENTOS: %d",
           soma(mat,tam));

    for (pp=mat,i=0; i<tam; i++,pp++){
        free(*pp);
    }

    free(mat);

    return 0;
}

int soma(int **mat, int tam){
    int **pp, *p, i,j,soma=0;
    for (pp=mat,i=0; i<tam; i++,pp++){
        for (p=*pp,j=0; j<tam; j++,p++){
            soma += (*p);
        }
    }
    return soma;
}
```

Para treinar

Refazer os exemplos 2 e 4 sem utilizar indexação

4. Tópicos Relevantes – Conteúdo

Tópicos Relevantes

Argumentos

Parâmetros Formais da Função `main`

- A função **`main`** recebe dois parâmetros:
  - **`argc`**, um parâmetro do tipo inteiro que possui o número de argumentos passados
  - **`argv`**, um parâmetro do tipo arranjo para arranjo de caracteres que pode ser entendido como um arranjo de *strings*
- É importante mencionar que o parâmetro **`argc`** será, no mínimo, de tamanho `1`, pois o **`argv[0]`** é sempre o nome do aplicativo

Observe o aplicativo `show` abaixo:

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    int i;
    printf("Numero de argumentos passados: %d\n", argc);

    for (i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }

    return 0;
}
```

Ao invocar o aplicativo `show` pelo *prompt* de comando, exibir-se-á:

- ./show
  - `Numero de argumentos passados:  1`
  - `argv[0] = ./show`
- ./show 2 5.7
  - `Numero de argumentos passados:  3`
  - `argv[0] = ./show`
  - `argv[1] = 2`
  - `argv[2] = 5.7`

Delimitador

- O espaço é o delimitador dos argumentos. Contudo, se for necessário inserir um parâmetro que possua espaço, então deverá ser utilizada aspas. Observe:

- ./show "Ricardo Terra" "Linguagem C" 2015
  - **Numero de argumentos passados: 4**
  - **argv[0] = ./show**
  - **argv[1] = Ricardo Terra**
  - **argv[2] = Linguagem C**
  - **argv[3] = 2015**

Aplicabilidade

- A partir de agora sabe-se que os argumentos são úteis para quando se deseja passar informações ao aplicativo no momento de sua chamada

- Como todo argumento é um *string* existem algumas funções de conversão da biblioteca **`stdlib.h`** muito utilizadas:
  - **`int atoi(char *)`**: converte um *string* para **`int`**
  - **`long int atol(char *)`**: converte um *string* para **`long int`**
  - **`float atof(char *)`**: converte um *string* para **`float`**

Exemplo Completo: Criando programa `soma`

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

int main(int argc, char *argv[]) {
    float x, y;

    if (argc != 3) {
        printf("Uso: soma <n1> <n2>");
        return -2;
    }

    x = atof(argv[1]);
    y = atof(argv[2]);

    printf("%.2f", x + y);

    return 0;
}
```

Para treinar

Estender o programa **soma** de forma que possa se passar infinitos números, por exemplo:

```
./soma 3.2 8.1 4.2 1.5
```

Tópicos Relevantes

União

União

- Em C, uma **union** é uma posição de memória compartilhada por duas ou mais variáveis, normalmente de tipos diferentes

União

- A definição de uma **union** é semelhante à definição de uma estrutura, conforme pode ser observado a seguir:

```
typedef union {
    tipo nome;
    tipo nome;
    tipo nome;
} nomeUniao;
```

O exemplo abaixo cria uma união que armazena um inteiro `i` e um caractere `ch` compartilhando uma mesma posição de memória em uma união chamada `intChar`:

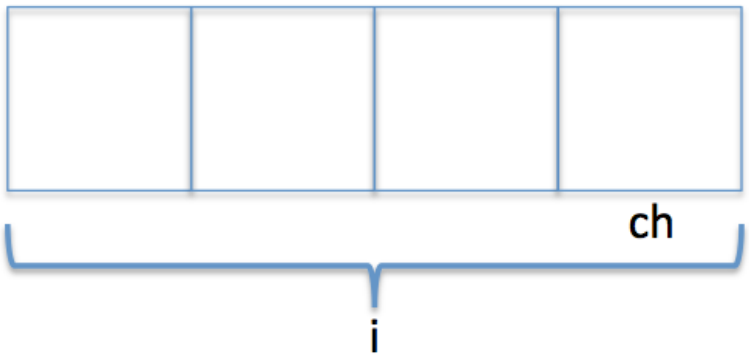
```
1 typedef union {
      int i;
3     char ch;
  } intChar;
```

Para declarar uma variável do tipo `intChar`, faz-se como abaixo:

- `intChar x;`

União

- Na união **`intChar`**, o inteiro **`i`** e o caractere **`ch`** compartilham a mesma posição de memória
- A união terá o tamanho de um inteiro, uma vez que **`int`** é o maior tipo de variável da união
- Desse modo, a variável **`i`** ocupa 4 *bytes* e a variável **`ch`** compartilha 1 *byte* – o mais à direita

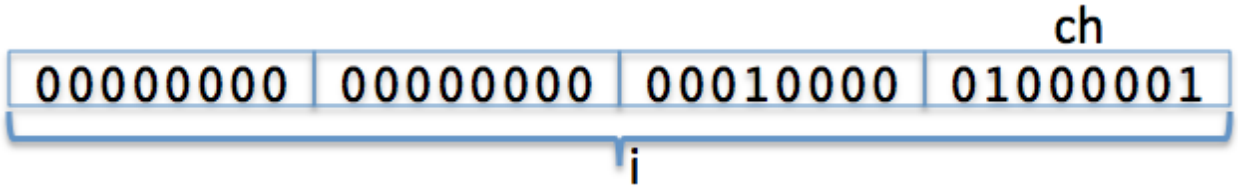


União

- O acesso a elementos de uma união é similar ao modo que se faz em estruturas
- Em variáveis do tipo da união utiliza-se o operador ponto (.)
- Em ponteiros para o tipo da união utiliza-se o operador seta (->)

Exemplo 1: Atribuição e leitura de campos da união `intChar`

```
#include<stdio.h>

typedef union {
    int i;
    char ch;
} intChar;

int main(int argc, char *argv[]) {
    intChar x;

    x.i = 4161;

    printf("%d", x.i); /* Imprime 4161 */
    printf("%c", x.ch); /* Imprime 'A' */

    return 0;
}
```

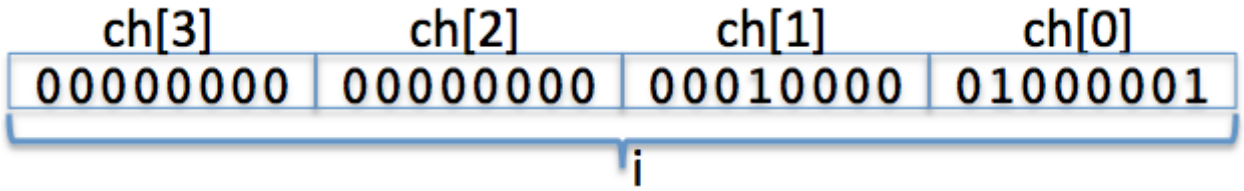


Exemplo 2: Atribuição e leitura de campos da união `intChar` por meio de ponteiros

```
1  #include<stdio.h>

3  typedef union {
       int i;
5      char ch;
   } intChar;
7
   void le(intChar *p) {
9      scanf("%d", &(p->i));
   }
11
   void imprime(intChar *p) {
13     printf("%d %c", p->i, p->ch); /* Imprime 4161 A sse digitou 4161 */
   }
15
   int main(int argc, char *argv[]) {
17     intChar x;

19     le(&x);
       imprime(&x);
21
       return 0;
23 }
```

Exemplo 3: Definição da união `pw`

```
1  #include<stdio.h>
2
3  typedef union {
4      int i;
5      char ch[4];
6  } pw;
7
8  int main(int argc, char *argv[]) {
9      pw senha;
10     senha.ch[0] = 65;
11     senha.ch[1] = 16;
12     senha.ch[2] = senha.ch[3] = 0;
13
14     printf("%d", senha.i); /* Imprime 4161 */
15
16     return 0;
17 }
```



Tópicos Relevantes

Enumeração

Enumeração

- Uma **enum** é um conjunto de constantes inteiras que determina quais os possíveis valores de uma variável desse tipo

Sintaxe:

- `typedef enum {x, y, z} exemplo;`

- Por padrão, os valores começam de zero. Por exemplo, na enumeração definida acima, as variáveis possuem os seguintes valores:
  - `x = 0`
  - `y = 1`
  - `z = 2`

Alterando o valor das variáveis de uma enumeração

- Contudo, as variáveis podem ter sua numeração alterada com uma simples atribuição no momento da definição. Por exemplo:

- `typedef enum {x, y=10, z} exemplo;`

- Assim, as variáveis possuem os seguintes valores:
  - `x = 0`
  - `y = 10`
  - `z = 11`

Exemplo 1: Definindo o tipo `boolean`

```
1  #include<stdio.h>

3  typedef enum {false, true} boolean;

5  int main(int argc, char *argv[]) {
       boolean b;
7
       b = (3 > 1);
9      if (b) {
           printf("3 > 1");
11     }

13     b = !b;

15     if (b == true) {
           printf("3 <= 1");
17     }

19     return 0;
   }
```

Exemplo 2: Definindo o tipo `diaSemana`

```c
#include<stdio.h>

typedef enum {
    domingo, segunda, terca, quarta, quinta, sexta, sabado
} diaSemana;

int main(int argc, char *argv[]) {
    diaSemana d = segunda; /* Poderia ser 1 */

    if (d == sabado || d == domingo) {
        printf("Fim de Semana!");
    } else {
        printf("Dia Util.");
    }

    return 0;
}
```

Observações Importantes:

- Observe que **segunda** refere-se ao inteiro **1**
  - `d = segunda` equivale a `d = 1`
- Isso indica que **segunda** não é o *string* "`segunda`"
  - `d = segunda` **não equivale a** `d = "segunda"`
- Assim, caso o usuário digite "`segunda`" e você quer atribuir isso à variável **d**, deve ser feito com alguns comandos `if`

```
1 if (!strcmp(str,"segunda")){
      d = segunda;
3 } else if (!strcmp(str,"terça")){
      d = terca;
5 } ...
```

Tópicos Relevantes

Diretivas

Pré-Processador de C

- O pré-processador manipula o texto de um programa **antes da compilação**
- Pode-se incluir diversas instruções do compilador no código fonte de um programa C. Essas instruções são denominadas **diretivas**
- Basicamente, diretivas são linhas cujo primeiro caractere é um # (sustenido)

O pré-processador de C reconhece as seguintes diretivas:

`#include`	`#if`
`#define`	`#ifdef`
`#undef`	`#ifndef`
`#line`	`#else`
`#error`	`#elif`
`#pragma`	`#endif`

Diretiva `#include`

- Instrui o compilador a inserir todo o conteúdo de outro arquivo no arquivo fonte

Sintaxe:

- `#include<stdio.h>`
  - com < ... > , delimita-se arquivos da biblioteca C
- `#include "mylib.h"`
  - com "..." (aspas), delimita-se arquivos fora da biblioteca C, ou melhor, aqueles criados por você

Diretiva `#include` – Exemplo

```
1  #include<stdio.h>
   #include"mylib.h"
3
   int main(int argc, char *argv[]) {
5      int x = 2, y = 3;
       printf("soma: %d", soma(x, y));
7      return 0;
   }
9
   int soma(int x, int y) {
11     return x + y;
   }
```

Arquivo `mylib.h`

```
   /* Prototipos */
2  int soma(int, int);
```

Diretiva `#define`

- Define uma macro – um *string* de substituição. O Pré-Processador de C substituirá cada ocorrência dessa macro pela seu valor **antes** da compilação

Exemplos:

- ```
  #define PI 3.14159
  area = PI * pow(r,2); ↦ area = 3.14159 * pow(r,2);
  ```

- ```
  #define TAM_MAX 100
  int = v[TAM_MAX]; ↦ int v[100];
  ```

- ```
  #define escreve printf
  escreve("%d",i); ↦ printf("%d",i);
  ```
  ```

Diretiva `#define` – Exemplo

```
1  #include<stdio.h>
   #include<math.h>
3
   #define PI 3.14159
5  #define escreve printf

7  int main(int argc, char *argv[]) {
       double r;
9      scanf("%lf", &r);

11     escreve("%lf", PI*pow(r, 2)); /*printf("%lf", 3.14159*pow(r,2));*/

13     return 0;
   }
```

Diretiva `#undef`

- Uma macro pode ser removida com a diretiva **`undef`** em qualquer ponto do arquivo fonte, isto é, a macro só vale em uma parte do código fonte

Por exemplo, removendo as macros definidas anteriormente:

- `#undef PI`
- `#undef TAM_MAX`
- `#undef escreve`

Diretiva `#if`, `#endif`, `#elif` e `#else`

- Permitem fazer pré-compilações condicionais

Exemplo

- Define-se uma macro **`modelo`** e dependendo de seu valor, define-se uma outra macro

```
#define modelo medio

#if modelo == pequeno
#define TAM 10
#elif modelo == medio
#define TAM 100
#else
#define TAM 1000
#endif
```

Diretiva `#ifdef` e `#ifndef`

- Permite fazer pré-compilações condicionais em função de uma macro ter sido (**`#ifdef`**) ou não definida (**`#ifndef`**)

Exemplo

- Define-se uma macro **`debug`** e, caso ela esteja definida, uma série de instruções voltadas ao teste do programa serão executadas

- Por exemplo:

```
1 #define debug;

3 #ifdef debug
      printf("%d",i);
5 #endif
```

Demais diretivas

- `#line`: Reseta o contador de linhas. Geralmente utilizado para depuração e aplicações específicas
- `#error`: Força o compilador a parar a compilação. Geralmente utilizado para depuração
- `#pragma`: Permite que sejam passadas instruções ao compilador
- Existem também os operadores `#` e `##`. Utilizados com a diretiva `define` quando essa diretiva visa definir algo parecido com uma função

Exemplo Completo

```c
#include<stdio.h>
#include"mylib.h"

#define LANG PT

#if LANG == PT
#define para    for
#define escreve printf
#define le      scanf
#elif LANG == EN
#define write   printf
#define read    scanf
#endif

#undef LANG

#define TAM 5
#define DEBUG
```

```c
int main(int argc, char *argv[]) {
    int v[TAM], i, *p, ix, iy;

    para (p = v, i = 0; i < TAM; i++, p++) {
        le("%d", p);
    }

#ifdef DEBUG
    para (p = v, i = 0; i < TAM; i++, p++) {
        escreve("v[%d]=%d\n", i, *p);
    }
#endif

    le("%d", &ix);
    le("%d", &iy);

#ifdef DEBUG
    escreve("%d -> %d e %d -> %d\n", ix, v[ix], iy, v[iy]);
#endif

    escreve("%d", soma(v[ix], v[iy]));
    return 0;
}

int soma(int x, int y) {
    return x + y;
}
```

Tópicos Relevantes

Arquivos

Funções de E/S

- As funções que compõem o sistema de entrada/saída do C ANSI podem ser agrupadas em duas principais categorias:
  - E/S pelo console e E/S com arquivo

- A linguagem C não contém nenhum comando de E/S. Ao contrário, todas as operações de E/S ocorrem mediante chamadas de funções da biblioteca C padrão (**`stdio.h`**)

Stream x Arquivo

- Antes de aprofundarmos em nas funções de E/S, é importante diferenciar *stream* e *arquivo*
- O sistema de E/S de C fornece uma interface consistente ao programador, independente do dispositivo real que é acessado
- Essa abstração é chamada *stream* e o dispositivo real é chamado de *arquivo*

Streams

- Existem dois tipos principais de *streams*:
  - *stream* de texto
  - *stream* binário

Stream de texto

- Basicamente, um *stream* de texto é uma sequência de caracteres
- Normalmente, um *stream* de texto é organizada em linhas terminadas por um caractere de nova linha

todo.txt

```
1.  Buscar Jornal
2.  Ir ao supermercado
3.  Marcar dentista
```

Stream de texto

- É importante mencionar que a representação de nova linha é diferente em sistemas operacionais
- Por exemplo, em sistemas operacionais baseados em UNIX é representado por:
  - \n – ASCII 10
  - isto é, somente a quebra de linha (*line feed*)
- Já no "*software*" Windows é representado por:
  - \r\n – ASCIIs 13 e 10
  - isto é, retorno de carro (*carrier return*) seguido de quebra de linha (*line feed*)

Stream de texto

- Até agora, temos trabalhado com as *streams* padrões:
  - `stdin` → para entrada de dados
  - `stdout` → para saída de dados

Stream **binário**

- Basicamente, um *stream* binário é uma sequência de *bytes* diretamente correspondida com o dispositivo externo, isto é, sem tradução dos caracteres

todo.bin

```
10010101010010101011101010100101010111010101001010
10101001010101001010101110101010010101011101010100
11111010100101010111010101001010101011101010100101
10100101010111010101001010101110101010010101100101
```

Arquivos

- Arquivo pode ser qualquer dispositivo de E/S. Por exemplo:
  - impressora, teclado, disquete, disco rígido etc
- Os aplicativos C ANSI manipulam arquivos através de *streams*
- Para manipular o ***stream*** de um **arquivo**, é necessário que o arquivo seja aberto
- **IMPORTANTE**: Após manipular o arquivo, esse **deve** ser fechado

Operações comuns em arquivos:

- abrir e fechar
- apagar
- ler e escrever caracteres ou dados binários
- verificar se chegou ao fim
- posicionar

Observações:

- Obviamente algumas dessas funções não se aplicam a todos os tipos de dispositivos. Por exemplo:
  - não é possível reposicionar uma impressora no início
  - um arquivo em disco permite acesso aleatório ao contrário de um teclado

Fechar arquivos

- **Sempre** após manipular um arquivo, ele **deve** ser fechado
- Isso faz com que os *buffers* sejam descarregados para o dispositivo externo, o que ocorre automaticamente quando o aplicativo é encerrado
- Entretanto, caso o aplicativo encerre de forma não esperada, o *buffer* do arquivo pode não ser descarregado, gerando uma inconsistência

Funções de Entrada e Saída (E/S)

- As funções de entrada e saída encontram-se na biblioteca `stdio.h`
- Por padrão, a maioria das funções iniciam-se com o prefixo "`f`"
- No próximo *slide*, serão vistas diversas dessas funções

Função	Descrição
`fopen()`	Abre um arquivo
`fclose()`	Fecha um arquivo
`fputc()`, `putc()`	Escreve um caractere em um arquivo
`fgetc()`, `getc()`	Lê um caractere de um arquivo
`fprintf()`	Equivalente a `printf()`, utilizando *stream*
`fscanf()`	Equivalente a `scanf()`, utilizando *stream*
`fgets()`	Lê uma cadeia de caracteres (*string*)
`fputs()`	Escreve uma cadeia de caracteres (*string*)
`fseek()`	Posiciona o arquivo em um ponto específico
`rewind()`	Posiciona o arquivo no início
`feof()`	Verifica se chegou ao final do arquivo
`ferror()`	Verifica se ocorreu um erro
`fflush()`	Descarrega o *buffer* associado ao arquivo
`fread()`	Leitura binária de dados
`fwrite()`	Escrita binária de dados
`remove()`	Remove um arquivo

Ponteiro de Arquivo

- Para ter acesso aos dados em um arquivo é necessário definir um ponteiro do tipo especial **`FILE`** – definido na biblioteca **`stdio.h`**
- Esse ponteiro permite que o aplicativo tenha acesso a uma estrutura que armazena informações importantes sobre o arquivo
- Declaração:
  - `FILE *fp;`

    `fp` → ponteiro utilizado para operações no arquivo

Abertura de um Arquivo

- A primeira operação a ser realizada em um arquivo é a sua abertura. Essa operação associa um *stream* (fluxo de dados) ao arquivo

- Um arquivo pode ser aberto para diversas finalidades:
  - leitura, escrita, leitura/escrita, adição (`append`) de texto etc

Abertura de um Arquivo

- A função **fopen** é utilizada para abrir o arquivo
- Protótipo:

```
FILE *fopen (const char *arq, const char *modo)
```

  - **arq** → nome do arquivo (caminho relativo ou absoluto)
  - **modo** → determina como o arquivo será aberto
- Assim como na função **malloc**, o ponteiro retornado não deve ser modificado. Além disso, caso não consiga abrir o arquivo, é retornado um ponteiro nulo (`NULL`)
- No próximo *slide*, serão vistos os diversos modos de abertura de arquivos

Tópicos Relevantes – Arquivos

Modo	Descrição
`r`	Abre um arquivo texto para leitura
`w`	Cria um arquivo texto para escrita
`a`	Adiciona texto ao fim de um arquivo texto
`rb`	Abre um arquivo binário para leitura
`wb`	Abre um arquivo binário para escrita
`ab`	Anexa a um arquivo binário
`r+`	Abre um arquivo texto para leitura/escrita
`w+`	Cria um arquivo texto para leitura/escrita
`a+`	Anexa ou cria um arquivo texto para leitura/escrita
`r+b`	Abre um arquivo binário para leitura/escrita
`w+b`	Cria um arquivo binário para leitura/escrita
`a+b`	Anexa a um arquivo binário para leitura/escrita

Importante:

Se um arquivo **existente** for aberto para escrita, será apagado e um novo será criado. Se for aberto para leitura/escrita, ele não será apagado. E, caso ele não exista, então será criado

Exemplo: Abertura de um arquivo

```
FILE *fp; char ch;
fp = fopen("arquivo.txt", "r");
if (!fp) {
    printf("O arquivo nao pode ser aberto");
    return -3;
}
```

Importante:

Usualmente, testa-se o retorno da função **fopen** para verificar se houve erro na abertura de um arquivo, tal como arquivo não existente, arquivo sem permissão de leitura ou escrita etc

Fechamento de um Arquivo

- A função **`fclose`** fecha o *stream* aberto pela função **`fopen`**

Protótipo:

- `int fclose (FILE *fp)`
  - **`fp`** → ponteiro para o arquivo a ser fechado

Fechamento de um Arquivo

- Caso a função retorne `0` indica que o arquivo foi fechado com sucesso. Qualquer outro valor indica que ocorreu algum erro

- A chamada a função `fclose` implica na escrita de qualquer dado que ainda não tenha sido **efetivamente** escrito no arquivo
  - Isso é um ponto importante a mencionar, pois no UNIX, por exemplo, uma operação de escrita em arquivo não ocorre imediatamente à emissão da ordem de escrita

  - Normalmente, o sistema operacional executa a escrita no momento que achar mais conveniente

Verificando o Final de um Arquivo

- A função **feof** indica se chegou ao final do arquivo

Protótipo:

- `int feof (FILE *fp)`
  - **fp** → ponteiro do arquivo a ser verificado se atingiu o final

Exemplo

- O uso mais comum de **feof** é em um laço de repetição, por exemplo, para a leitura completa de um arquivo

```
1 while ( !feof(fp) ) { /* Poderia ser while( feof(fp) == 0 ) */
      ch = fgetc(fp); /* A função fgetc ou getc lê um caractere */
3     ...
  }
```

Retrocedendo um Arquivo ao Início

- A função **`rewind`** reposiciona o arquivo em seu início
  - Operação semelhante a de rebobinar um VHS

Protótipo:

- `void rewind(FILE *fp)`
  - **`fp`** → ponteiro para o arquivo a ser retrocedido

- É importante observar que o modo de abertura do arquivo
  - Por exemplo, um arquivo aberto somente para escrita, se retrocedido ao início, não irá permitir outra operação que não seja escrita

Leitura e Escrita de Caracteres

- As operações mais simples em arquivos são leitura e escrita de caracteres
- Para efetuar a leitura do caractere de um arquivo pode-se utilizar as funções **`getc`** e **`fgetc`**, que são equivalentes

Protótipo

- `int fgetc (FILE *fp)`
  - **`fp`** → ponteiro para o arquivo cujo caractere será lido
- A função retorna o caractere como um **`int`** (padrão)
- Caso tente-se ler de um arquivo cujo final já foi alcançado, a função devolve o código **`EOF`** (-1)

Leitura e Escrita de Caracteres

- Para efetuar a escrita de um caractere em um arquivo pode-se utilizar as funções **`putc`** e **`fputc`**, que são equivalentes

Protótipo

- `int fputc (int ch, FILE *fp)`
  - **`fp`** → ponteiro para o arquivo cujo caractere será escrito
  - **`ch`** → caractere a ser escrito

Exemplo 1a: Exibindo o conteúdo de um arquivo

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    FILE *fp; char ch;

    fp = fopen("arquivo.txt", "r");
    if (!fp) {
        printf("O arquivo nao pode ser aberto");
        return -3;
    }

    while (!feof(fp)) {
        ch = fgetc(fp);
        if (ch != EOF) {
            printf("%c", ch);
        }
    }
    fclose(fp);
    return 0;
}
```

Exemplo 1b: Criando um `cat` simplificado

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    FILE *fp; char ch;

    if (argc != 2) {
        printf("Uso: cat <nome-do-arquivo>");
        return -2;
    }

    fp = fopen(argv[1], "r");

    if (!fp) {
        printf("O arquivo nao pode ser aberto");
        return -3;
    }

    while (!feof(fp)) {
        ch = fgetc(fp);
        if (ch != EOF) {
            printf("%c", ch);
        }
    }

    fclose(fp);
    return 0;
}
```

Exemplo 1c: Criando o próprio `cat`

```c
#include<stdio.h>

int main(int argc, char *argv[]) {
    FILE *fp; char ch; int i;

    if (argc < 2) {
        printf("Uso: cat <nome-do-arquivo-1> ... <nome-do-arquivo-n>\n");
        return -2;
    }

    for (i=1; i < argc; i++){
        fp = fopen(argv[i], "r");

        if (!fp) {
            printf("O arquivo %s nao pode ser aberto", argv[i]);
            return -3;
        }

        while (!feof(fp)) {
            ch = fgetc(fp);
            if (ch != EOF) {
                printf("%c", ch);
            }
        }

        fclose(fp);
    }
    return 0;
}
```

Exemplo 2: Escrita, Retrocesso e Leitura

```c
#include<stdio.h>

int main(int argc, char *argv[]) {
    FILE* fp; int i; char ch;

    fp = fopen("alfabeto.txt", "w+"); /* Abre para leitura/escrita */
    if (!fp) {
        printf("Erro na abertura do arquivo.");
        return -3;
    }

    for (i = 65; i <= 90; i++) { /* Grava de A a Z */
        fputc(i, fp);
    }

    rewind(fp); /* Rebobina o arquivo */

    while (!feof(fp)) { /* Leitura Completa */
        ch = fgetc(fp);
        if (ch != EOF) {
            printf("%c", ch);
        }
    }

    fclose(fp); /* Fecha o arquivo */
    return 0;
}
```

Exemplo 3: Criando o próprio cp

```c
#include<stdio.h>

int main(int argc, char *argv[]) {
    FILE *fp1, *fp2; char ch;
    if (argc != 3) {
        printf("Uso: cp <arquivo-origem> <arquivo-destino>");
        return -2;
    }

    fp1 = fopen(argv[1], "r"); /* Abre para leitura */
    if (!fp1) {
        printf("Erro na abertura do arquivo origem.");
        return -3;
    }
    fp2 = fopen(argv[2], "w"); /* Abre para escrita */
    if (!fp2) {
        printf("Erro na abertura do arquivo destino.");
        return -3;
    }
    while (!feof(fp1)) { /* Leitura/Gravacao */
        ch = fgetc(fp1);
        if (ch != EOF) {
            fputc(ch, fp2);
        }
    }

    fclose(fp1); fclose(fp2);     /* Fecha os arquivos */
    return 0;
}
```

Leitura e Escrita de Cadeia de Caracteres (*strings*)

- As funções **`fgets`** e **`fputs`** realizam a leitura e escrita de cadeias de caracteres em arquivos

Protótipo `fputs`:

- `int fputs(char *str, FILE *fp)`
  - **`str`** → *string* a ser gravado
  - **`fp`** → ponteiro do arquivo cujo *string* será escrito
- A função **`fputs`** escreve o *string* **`str`** no *stream* **`fp`**
- Em caso de erro, a função **`fputs`** retorna **`EOF`**

Protótipo `fgets`:

- `int fgets(char *str, int comp, FILE *fp)`
  - **`str`** → *string* a ser preenchido pela leitura
  - **`comp`** → tamanho do *string*
  - **`fp`** → ponteiro do arquivo cuja *string* será lido
- A função **`fgets`** lê uma cadeia de caracteres do *stream* **`fp`** para a variável **`str`**
  - Até que um caractere de nova linha seja encontrado ou **`comp-1`** caracteres sejam lidos
- Em caso de erro, a variável **`str`** recebe **`NULL`**

Exemplo 4: Escrita, Retrocesso e Leitura utilizando `fgets` e `fputs`

```c
#include<stdio.h>

#define STR_TAM 100

int main(int argc, char *argv[]) {
    char linguagens[5][20] = { "PASCAL", "C", "C++", "SmallTalk", "Java" };
    FILE* fp; int i; char str[STR_TAM];

    fp = fopen("linguagens.txt", "w+"); /* Cria para leitura/escrita */
    if (!fp) {
        printf("Erro na abertura do arquivo.");
        return -3;
    }
    /* Grava o arranjo de linguagens */
    for (i = 0; i < sizeof(linguagens) / sizeof(linguagens[i]); i++) {
        fputs(linguagens[i], fp);
        fputc('\n', fp);
    }

    rewind(fp); /* Rebobina o arquivo */

    while (!feof(fp)) { /* Leitura Completa */
        if (fgets(str, STR_TAM, fp) != NULL) {
            printf("%s", str);
        }
    }
    fclose(fp); /* Fecha o arquivo */
    return 0;
}
```

E/S Formatada

- As funções **`fprintf`** e **`fscanf`** são equivalentes as já conhecidas funções **`printf`** e **`scanf`**
- As funções **`printf`** e **`scanf`** trabalham **sempre** com os arquivos padrões de E/S
  - **`stdin`**: entrada padrão, normalmente, teclado
  - **`stdout`**: saída padrão, normalmente, monitor
- A única diferença é que nas funções **`fprintf`** e **`fscanf`** deve-se informar qual o arquivo que está se trabalhando

Protótipo `fprintf`:

- `int fprintf(FILE *fp, const char *format, ...)`
  - **`fp`** → ponteiro do arquivo a ser escrito
  - **`format`** → sequência de conversão
  - `...` → variáveis (*varargs*)
- Por exemplo:
  `printf("%d", num)` ↔ `fprintf(stdout,"%d", num)`

Protótipo `fscanf`:

- `int fscanf(FILE *fp, const char *format, ...)`
  - **`fp`** → ponteiro do arquivo a ser escrito
  - **`format`** → sequência de conversão
  - `...` → variáveis (*varargs*)
- Por exemplo:
  `scanf("%d", &num)` ↔ `fscanf(stdin,"%d", &num)`

E/S Formatada

- Embora essas funções, por sua semelhança com `printf` e `scanf`, sejam maneiras convenientes de escrever e ler dados de arquivos, elas têm a desvantagem de serem mais lentas do que uso de arquivos binários, os quais são descritos a frente

- A perda de tempo ocorre devido aos dados serem armazenados na codificação ASCII, o que obriga conversão de dados a cada operação

- Contudo, dados armazenados em ASCII pode ser também uma vantagem, uma vez que são facilmente verificados pelos usuários, o que não ocorre com dados binários

Exemplo 5: Gravando uma matriz em um arquivo

```c
#include<stdio.h>

int main(int argc, char *argv[]) {
    FILE* fp;
    int i, j, tam, aux;

    printf("Tamanho da Matriz Quadrada: ");
    scanf("%d", &tam);

    fp = fopen("matriz.txt", "w"); /* Abre para escrita */
    if (!fp) {
        printf("Erro na abertura do arquivo.");
        return -3;
    }
    fprintf(fp, "%d\n", tam); /* Armazena o tamanho da matriz */

    for (i = 0; i < tam; i++) { /* Le matriz gravando no arquivo */
        for (j = 0; j < tam; j++) {
            printf("mat[%d][%d] = ", (i + 1), (j + 1));
            scanf("%d", &aux);
            fprintf(fp, "%d ", aux);
        }
        fprintf(fp, "\n");
    }

    fclose(fp); /* Fecha o arquivo */

    return 0;
}
```

Exemplo 6: Lendo uma matriz de um arquivo

```c
#include<stdio.h>

int main(int argc, char *argv[]) {
    FILE* fp;
    int i, j, tam, aux;

    fp = fopen("matriz.txt", "r"); /* Abre para leitura */

    if (!fp) {
        printf("Erro na abertura do arquivo.");
        return -3;
    }

    fscanf(fp, "%d", &tam); /* Le o tamanho */

    for (i = 0; i < tam; i++) { /* Imprime a Matriz Gravada no Arquivo */
        for (j = 0; j < tam; j++) {
            fscanf(fp, "%d", &aux);
            fprintf(stdout, "%d ", aux);
        }
        fprintf(stdout, "\n");
    }

    fclose(fp); /* Fecha o arquivo */

    return 0;
}
```

Exercício de Fixação

Refazer os Exemplos 5 e 6 com matrizes que não sejam quadradas

Leitura e Escrita Binária

- As funções **`fread`** e **`fwrite`** são empregadas para leitura e escrita de dados binários

Protótipo `fread`:

- `size_t fread (void *ptr, size_t size, size_t nmemb, FILE *fp)`
  - **`ptr`** → *ponteiro para* ou *endereço* do tipo de dado a ser atribuído
  - **`size`** → tamanho do tipo de dado
  - **`nmemb`** → número de elementos a serem lidos
  - **`fp`** → ponteiro para o arquivo a ler
- Lê **`nmemb`** membros, cada um com **`size`** *bytes* do arquivo **`fp`** e os coloca em **`ptr`**
- Retorna o número de itens que foram lidos com sucesso
- Em caso de erro ou o fim de arquivo, o valor de retorno é menor do que **`nmemb`** ou zero
- Como essa função não distingue entre um fim de arquivo e erro, é aconselhável o uso de **`feof`** ou **`ferror`** para determinar o problema

Protótipo `fwrite`:

- `size_t fwrite (const void *ptr, size_t size, size_t nmemb, FILE *fp)`
  - **`ptr`** → *ponteiro para* ou *endereço* o dado a ser gravado
  - **`size`** → tamanho do tipo de dado
  - **`nmemb`** → número de elementos a serem gravados
  - **`fp`** → ponteiro para o arquivo a gravar
- Escreve **`nmemb`** membros, cada um com **`size`** *bytes* a partir do endereço apontado em **`ptr`** para o arquivo **`fp`**
- Retorna o número de itens que foram escritos com sucesso
- Em caso de erro ou o fim de arquivo, o valor de retorno é menor do que **`nmemb`** ou zero

Exemplo 7a: Gravando uma estrutura em um arquivo

```c
#include<stdio.h>

typedef struct {
    int matricula;
    char nome[60];
} aluno;

int main(int argc, char *argv[]) {
    FILE* fp;
    aluno a = { 1, "Ricardo Terra" };

    fp = fopen("aluno.bin", "wb"); /* Abre para escrita */
    if (!fp) {
        printf("Erro na abertura do arquivo.");
        return -3;
    }

    fwrite(&a, sizeof(aluno), 1, fp); /* Grava a estrutura */

    fclose(fp);

    return 0;
}
```

Exemplo 8a: Lendo uma estrutura de um arquivo

```c
#include<stdio.h>

typedef struct {
    int matricula;
    char nome[60];
} aluno;

int main(int argc, char *argv[]) {
    FILE* fp;
    aluno a;

    fp = fopen("aluno.bin", "rb"); /* Abre para leitura */
    if (!fp) {
        printf("Erro na abertura do arquivo.");
        return -3;
    }

    fread(&a, sizeof(aluno), 1, fp); /* Le a estrutura */

    printf("A matricula %d pertence ao aluno %s", a.matricula, a.nome);

    fclose(fp);

    return 0;
}
```

Exemplo 7b: Gravando um arranjo de estrutura em um arquivo

```c
#include<stdio.h>

typedef struct {
    int matricula;
    char nome[60];
} aluno;

int main(int argc, char *argv[]) {
    FILE* fp;
    aluno v[3] = {{1,"Ricardo Terra"},
        {2,"Virgilio Borges"},
        {3,"Marco Tulio"}};

    int tam = sizeof(v) / sizeof(aluno);

    fp = fopen("aluno.bin", "wb"); /* Abre para escrita */
    if (!fp) {
        printf("Erro na abertura do arquivo.");
        return -3;
    }

    fwrite(v, sizeof(aluno), tam, fp); /* Grava o arranjo de estrutura */

    fclose(fp);

    return 0;
}
```

Exemplo 8b: Lendo várias estruturas de um arquivo

```c
#include<stdio.h>

typedef struct {
    int matricula;
    char nome[60];
} aluno;

int main(int argc, char *argv[]) {
    FILE* fp;
    aluno a;

    fp = fopen("aluno.bin", "rb"); /* Abre para leitura */
    if (!fp) {
        printf("Erro na abertura do arquivo.");
        return -3;
    }

    while (!feof(fp)) {
        if (fread(&a, sizeof(aluno), 1, fp)) { /* Le a estrutura */
            printf("A matricula %d pertence ao aluno %s\n", a.matricula, a.nome);
        }
    }

    fclose(fp);

    return 0;
}
```

Exemplo 9: Criando o `cadastraAluno`

```
1  #include<stdio.h>
   #include<stdlib.h>
3  #include<string.h>

5  typedef struct {
       int matricula;
7      char nome[60];
   } aluno;
9
   int main(int argc, char *argv[]) {
11     FILE* fp; aluno a;

13     if (argc != 3) {
           printf("Uso: cadastra <matricula> \"<nome>\"\n");
15         return -2;
       }
17     fp = fopen("aluno.bin", "a+b"); /* Abre para escrita ao final */
       if (!fp) {
19         printf("Erro na abertura do arquivo.");
           return -3;
21     }
       a.matricula = atoi(argv[1]);
23     strcpy(a.nome,argv[2]);

25     fwrite(&a, sizeof(aluno), 1, fp);

27     fclose(fp);
       return 0;
29 }
```

5. Extras – Conteúdo

Extras

Função `printf`

A função `printf`

- A função `printf` é parte de um conjunto de funções pré-definidas armazenadas em uma biblioteca padrão de rotinas da linguagem C

- Ela permite apresentar na tela os valores de qualquer tipo de dado. Para tanto, `printf` utiliza o mecanismo de formatação, que permite traduzir a representação interna de variáveis para a representação ASCII que pode ser apresentada na tela

A função `printf`

- O primeiro argumento de `printf` é um string de controle, uma sequência de caracteres entre aspas. Essa string, que sempre deve estar presente, pode especificar através de caracteres especiais (as sequências de conversão) quantos outros argumentos estarão presentes nesta invocação da função

- Estes outros argumentos serão variáveis cujos valores serão formatados e apresentados na tela

A função `printf`

- Por exemplo, se o valor de uma variável inteira `x` é 12, então a execução da função:
  - `printf("Valor de x = %d", x);`
- Imprime na tela a frase: `Valor de x = 12`

A função `printf`

- Se y é uma variável do tipo caractere com valor 'A', então a execução de:
  - `printf("x = %d e y = %c \n", x, y);`
- Imprime na tela a frase: `x = 12 e y = A` seguida pelo caracter de nova linha (\n) , ou seja, a próxima saída para a tela aconteceria na linha seguinte
- Observe que a sequência de conversão pode ocorrer dentro de qualquer posição dentro da string de controle

A função `printf`

- A função `printf` não tem um número fixo de argumentos
- Em sua forma mais simples, pelo menos um argumento deve estar presente – *a string de controle*.
- Uma string de controle sem nenhuma sequência de conversão será literalmente impressa na tela
  - `printf("Estou aprendendo C ANSI!");`

A função printf

- Com variáveis adicionais, a única forma de saber qual o número de variáveis que será apresentado é por inspeção da string de controle
    - Desta forma, cuidado deve ser tomado para que o número de variáveis após a string de controle esteja de acordo com o número de sequências de conversão presente na string de controle

A função `printf`

- Além de ter o número correto de argumentos e sequências de conversão, o tipo de cada variável deve estar de acordo com a sequência de conversão especificada na string de controle

- A sequência de conversão pode ser reconhecida dentro da string de controle por iniciar sempre com o caractere %

As principais sequências de conversão para variáveis caracteres e inteiras são:

%c	Caractere simples
%d	Inteiro
%i	Lê um decimal inteiro (não pode ser octal ou hexadecimal)
%u	Lê um decimal sem sinal
%e	Notação científica
%f	Ponto flutuante
%g	%e ou %f (escolherá o mais curto)
%o	Octal
%s	Cadeia de caracteres (string)
%u	Inteiro sem sinal
%x	Hexadecimal
%p	Endereço de Memória

Os principais códigos especiais são:

- `\n` Nova linha
- `\t` Tabulação
- `\"` Aspas
- `\'` Apóstrofe
- `\\` Barra invertida
- `\0` Nulo

Exemplo Completo

```
1 #include<stdio.h>

3 int main(int argc, char *argv[]) {
      char c = 'A';
5     int i = 12;
      float f = 1.1;
7     double d = 3.1234;

9     printf("%c - %d - %f - %lf", c, i, f, d);

11    return 0;
  }
```

- Neste exemplo são impressas variáveis de todos os tipos primitivos da linguagem C. Observe a saída:
  `A - 12 - 1.100000 - 3.123400`

Exemplo Completo Customizado

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    char c = 'A';
    int i = 12;
    float f = 1.1;
    double d = 3.1234;

    printf("%c - %d - %.2f - %.4lf", c, i, f, d);
    return 0;
}
```

- Como o exemplo anterior, neste exemplo são impressas variáveis de todos os tipos primitivos da linguagem C, contudo houve a formatação dos tipos `float` e `double` para exibir somente 2 e 4 casas decimais, respectivamente. Observe a saída:

  `A - 12 - 1.10 - 3.1234`

Resumo

A função `printf` é parte de um conjunto de funções pré-definidas armazenadas em uma biblioteca padrão de rotinas da linguagem C. Ela permite apresentar na tela os valores de qualquer tipo de dado

Exemplo

```
int x = 5;

printf("Valor de x = %d", x);
```

Exemplo

O exemplo anterior é foi voltado à uma variável inteiro, contudo a mesma ideia pode ser seguida com os outros tipos primitivos

```
char c = 'A';
printf("Valor de c = %c", c);

float f = 3.14;
printf("Valor de f = %f", f);

double d = 9.290261;
printf("Valor de d = %.2lf", d);
```

Extras

Função `scanf`

A FUNÇÃO scanf

- A função scanf é uma das funções de entrada de dados da Linguagem C, que pode ser usada para ler virtualmente qualquer tipo de dado inserido por meio do teclado, frequentemente ela é usada para a entrada de números inteiros ou de ponto flutuante

A forma geral da função *scanf* é:

- `scanf("string de controle", lista de argumentos);`

A FUNÇÃO scanf

- Os especificadores de formato de entrada são precedidos por um sinal % e dizem à função scanf qual tipo de dado deve ser lido em seguida

- Esses códigos são listados na tabela a seguir

Código	Significado
%c	Lê um único caractere
%d	Lê um decimal inteiro
%i	Lê um decimal inteiro (não pode ser octal ou hexadecimal)
%u	Lê um decimal sem sinal
%e	Lê um número em ponto flutuante com sinal opcional
%f	Lê um número em ponto flutuante com ponto opcional
%g	Lê um número em ponto flutuante com expoente opcional (double)
%o	Lê um número em base octal
%s	Lê uma string
%x	Lê um número em base hexadecimal
%p	Lê um ponteiro

Observe alguns exemplos:

- Espera que o usuário digite um inteiro. O valor digitado será o conteúdo da variável `n`
  - `scanf("%d", &n);`

- Espera que o usuário digite um inteiro, um espaço e um ponto flutuante. O primeiro valor digitado será o conteúdo da variável m e o segundo valor será o conteúdo da variável `n`
  - `scanf("%d %f", &m, &n);`

Extras – Função `scanf`

- Foi refeito o exemplo do `printf` de forma que o usuário digite os valores a serem impressos

```
1  #include<stdio.h>

3  int main(int argc, char *argv[]) {
       char c; int i; float f; double d;
5
       printf("Digite um caractere:");
7      scanf("%c", &c);
       printf("Digite um inteiro:");
9      scanf("%d", &i);
       printf("Digite um float:");
11     scanf("%f", &f);
       printf("Digite um double:");
13     scanf("%lf", &d);

15     printf("%c - %d - %.2f - %.4lf", c, i, f, d);

17     return 0;
   }
```

Extras – Função `scanf`

- Você pode ler todas as variáveis em um único `scanf`

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    char c; int i; float f; double d;

    printf("Digite o caractere, inteiro, float e double:");
    scanf("%c %d %f %lf", &c, &i, &f, &d);

    printf("%c - %d - %.2f - %.4lf", c, i, f, d);

    return 0;
}
```

Extras – Função `scanf`

- O usuário deve digitar `n` números conforme lidos e depois será exibido a soma de todos eles. Note que o `printf` tem como finalidade somente orientar o usuário para a digitação dos números

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    int n, num, soma, i;
    printf("Digite a quantidade de numeros a serem somados:");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("Digite o numero %d/%d: ", i + 1, n);
        scanf("%d", &num);
        soma += num;
    }
    printf("Soma: %d", soma);
    return 0;
}
```

Resumo

A função `scanf` é uma das funções de entrada de dados da Linguagem C, que pode ser usada para ler virtualmente qualquer tipo de dado inserido por meio do teclado
Frequentemente, ela é usada para a entrada de números inteiros ou de ponto flutuante

Exemplo

```
int x;
printf("Digite o valor de x:  ");
scanf("%d", &x);
```

Isso ocorre para leitura de variáveis inteiras (`x` é inteiro). Para outros tipos de variáveis como `char`, `float` e `double` basta alterar `%d` para `%c`, `%f` e `%lf`, respectivamente

Extras

Funções para manipular *strings*

Biblioteca `string.h`

- A biblioteca **`string.h`** contém diversas funções para manipulação de *strings*. Por exemplo:
  - Comparar se dois *strings* são iguais
  - Copiar um *string* para dentro de outro *string*

Eis as funções que veremos:

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

void strcpy(char *strDestino, char *strOrigem);

void strcat(char *strDestino, char *str);
```

A função `strcmp`

- A função **`strcmp`** compara se dois *strings* são iguais. Como resultado tem-se:
  - Se forem iguais, retorna **`0`**
  - Se **`str1`** for maior que **`str2`**, retorna um valor positivo (**`>0`**)
  - Se **`str1`** for menor que **`str2`**, retorna um valor negativo (**`<0`**)

Exemplo: comparando os *strings* `str1` e `str2`

```
1  #include<stdio.h>
   #include<string.h>
3
   int main(int argc, char *argv[]) {
5      char str1[] = "Joao";
       char str2[] = "Carlos";
7
       if (strcmp(str1, str2) == 0) { /* sao iguais? */
9          printf("%s = %s", str1, str2);
       } else {
11         printf("%s != %s", str1, str2);
       }
13
       return 0;
15 }
```

Sendo um pouco mais profissional...

```
1  #include<stdio.h>
   #include<string.h>
3
   int main(int argc, char *argv[]) {
5      char str1[] = "Joao";
       char str2[] = "Carlos";
7
       if (!strcmp(str1, str2)) { /* sao iguais? */
9          printf("%s = %s", str1, str2);
       } else {
11         printf("%s != %s", str1, str2);
       }
13
       return 0;
15 }
```

A função `strcpy`

- A função **`strcpy`** copia para um *string* o conteúdo de um outro *string*

Exemplo

```
1  #include<stdio.h>
   #include<string.h>
3
   int main(int argc, char *argv[]) {
5      char str1[60], str2[60];

7      gets(str1); /* le o str1 */

9      strcpy(str2, str1); /* copia str1 para str2 */

11     printf("str1: %s\nstr2: %s", str1, str2);

13     return 0;
   }
```

A função `strcat`

- A função **`strcat`** concatena para um *string* o conteúdo de um outro *string*

Exemplo

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

int main(int argc, char *argv[]) {
    char str[60];

    strcpy(str, "rterrabh");
    strcat(str, "@gmail");
    strcat(str, ".com");

    printf("str: %s", str);

    return 0;
}
```

Extras

Outras funções relevantes

Biblioteca `time.h`

- Basicamente, a biblioteca **`time.h`** contém funções relativas a data/hora e medição de tempo

Eis as funções que veremos:

```
clock_t clock ( );

time_t time ( time_t* timer );
```

A função `clock`

- A função **`clock`** retorna o número de pulsos de *clock* decorrido desde o início da execução do programa

- Normalmente, utilizada para se medir o tempo de execução de um programa:
  - Faz-se isso armazenando o número de pulsos de *clock* no início e no final do programa e então se divide pelo macro **`CLOCKS_PER_SEC`** que armazena o número de pulsos de *clock* por segundo

Exemplo: medindo o tempo de execução do *fibonacci*

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

long int fib(int n) {
    if (n == 0 || n == 1) { /* Ponto de Parada */
        return n;
    }
    return fib(n - 1) + fib(n - 2); /* Codigo da Recursao */
}

int main(int argc, char *argv[]) {
    clock_t start = clock();
    int n;

    scanf("%d", &n);
    printf("fib(%d) = %ld\n", n, fib(n));

    printf("TEMPO: %fs", ((double) (clock() - start)) / CLOCKS_PER_SEC);

    return 0;
}
```

A função `time`

- A função **`time`** retorna o número de segundos a partir de 01/01/1970

Exemplo

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

int main(int argc, char *argv[]) {
    time_t agora;

    agora = time(NULL);
    printf("%ld segundos desde 01/01/1970\n", agora);
    printf("%ld horas desde 01/01/1970\n", agora / 3600);
    printf("%ld dias desde 01/01/1970\n", agora / (3600 * 24));

    return 0;
}
```

Biblioteca `ctype.h`

- Basicamente, a biblioteca **`ctype.h`** contém funções relativas a manipulação de caracteres

Eis as funções que veremos:

```
int isalnum ( int c ); /* alfanumerico? */

int isalpha ( int c ); /* alfabetico? */

int isdigit ( int c ); /* digito? */

int islower ( int c ); /* caixa baixa? */

int isupper ( int c ); /* caixa alta? */

int tolower ( int c ); /* converte para caixa baixa */

int toupper ( int c ); /* converte para caixa alta */
```

Exemplo `ctype.h`

```
1  #include <stdio.h>
   #include <ctype.h>
3
   int main(int argc, char *argv[]) {
5      char ch;
       scanf("%c", &ch);
7
       printf("Alfanumerico? %s\n", ((isalnum(ch)) ? "sim" : "nao"));
9      printf("Alfabetico? %s\n", ((isalpha(ch)) ? "sim" : "nao"));
       printf("Digito? %s\n", ((isdigit(ch)) ? "sim" : "nao"));
11     printf("Caixa baixa? %s\n", ((islower(ch)) ? "sim" : "nao"));
       printf("Caixa alta? %s\n", ((isupper(ch)) ? "sim" : "nao"));
13     printf("%c para %c\n", ch, tolower(ch));
       printf("%c para %c\n", ch, toupper(ch));
15
       return 0;
17 }
```

Biblioteca `math.h`

- Basicamente, a biblioteca **`math.h`** contém funções matemáticas

Eis as funções que veremos:

```
double log ( double x ); /* logaritmo base natural (ln) */

double log10 ( double x ); /* logaritmo base 10 */

double pow ( double x, double y ); /* exponenciação */

double sqrt ( double x ); /* raiz quadrada */

double floor ( double x ); /* arredonda para baixo */

double ceil ( double x ); /* arredonda para cima */

/* E outras comuns, tais como cos, sin, tan... */
```

Exemplo `math.h`

```
1  #include <stdio.h>
   #include <math.h>
3
   int main(int argc, char *argv[]) {
5      double n;
       scanf("%lf", &n);
7
       printf("ln(%lf) = %lf\n", n, log(n));
9
       printf("log10(%lf) = %lf\n", n, log10(n));
11
       printf("%lf ao quadrado = %lf\n", n, pow(n, 2));
13
       printf("raiz quadrada de %lf = %lf\n", n, sqrt(n));
15
       printf("%lf arrend. para baixo = %lf\n", n, floor(n));
17
       printf("%lf arrend. para cima = %lf\n", n, ceil(n));
19
       return 0;
21 }
```

Biblioteca `stdlib.h`

- Basicamente, a biblioteca **`stblib.h`** contém funções de propósito geral, tais como conversão, geração de números aleatórios etc

Eis as funções que veremos:

```
int atoi ( const char * str ); /* converte string para int */

double atof ( const char * str );  /* converte string para double */

long int atol ( const char * str );  /* converte string para long */

int rand ( void );  /* gera um numero aleatorio */

void srand ( unsigned int seed );  /* semente da aleatoriedade */

/* E outras já vistas, tais como malloc, free... */
```

Exemplo `stdlib.h`

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

int main(int argc, char *argv[]) {
    int numSecreto, numChute, limiteSuperior;
    if (argc != 2) {
        printf("Uso: adivinha <limiteSuperior>");
        return -2;
    }

    srand(time(NULL)); /* Inicializa a semente */
    limiteSuperior = atoi(argv[1]); /* Converte string para inteiro */
    numSecreto = (rand() % limiteSuperior) + 1; /* Gera numero secreto */

    do {
        printf("Adivinhe o numero (1 ate %d): ", limiteSuperior);
        scanf("%d", &numChute);
        if (numSecreto < numChute) {
            printf("Menor\n");
        } else {
            printf("Maior\n");
        }
    } while (numSecreto != numChute);
    printf("Parabens!");

    return 0;
}
```

Agradecimentos

Gostaria de agradecer ao aluno Willer Henrique dos Reis pela tarefa de conversão inicial dos *slides* em formato *ppt* para LaTeX

Referências


Victorine Viviane Mizrahi.
Treinamento em Linguagem C.
Prentice-Hall, 2 edition, 2008.


Herbert Schildt.
C: The Complete Reference.
McGraw-Hill, 4 edition, 2000.