

# Módulos



**Engenharia da Computação**

[www.eComp.Poli.br](http://www.eComp.Poli.br)

# Linguagem de Programação Imperativa (LPI)

**Prof. Joabe Jesus**

[joabe@ecomp.poli.br](mailto:joabe@ecomp.poli.br)

Linguagem C  
- Conceitos e Fundamentos  
Ferramentas

# MÓDULO 2

# Linguagem C

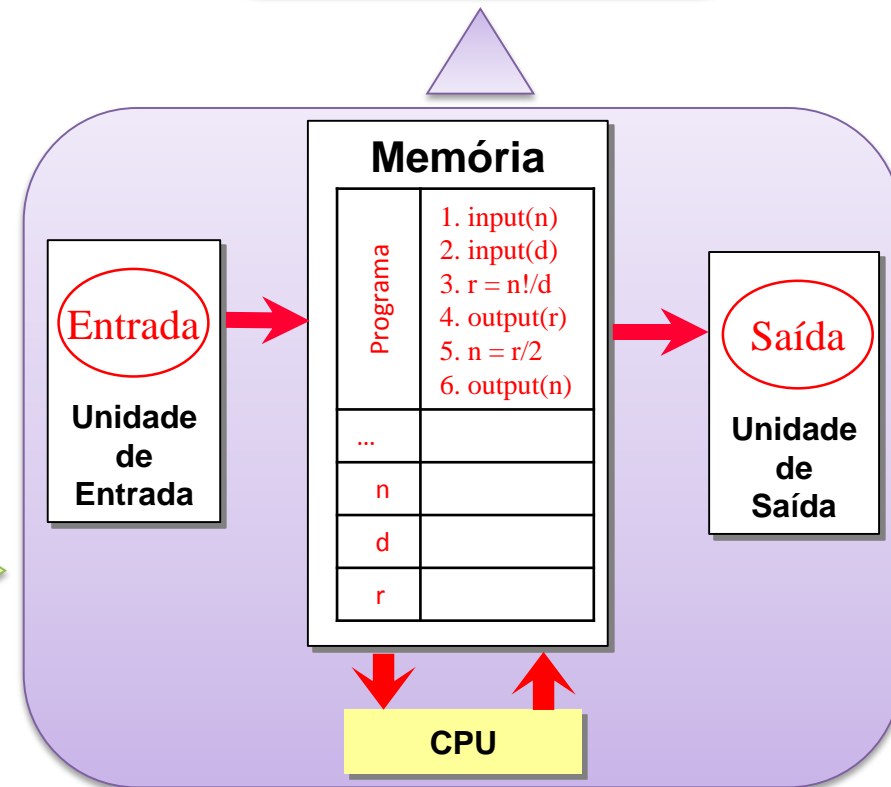
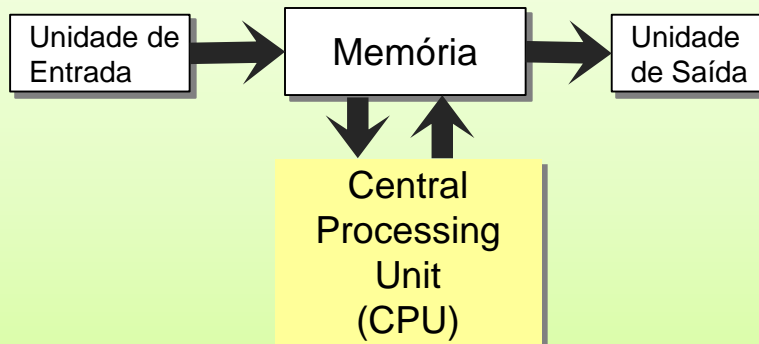
- Linguagem de programação **imperativa** bastante popular
  - Nível de dificuldade: médio
  - Ótima para aprender a programar
  - Programa diz o que deve fazer através de comandos que mudam o **estado da memória**
- Criada em 1972 (Dennis Ritchie)
  - Usada para escrever o UNIX
  - Muitas versões, em 1983 criou-se comitê de **padronização** ANSI
- Eficiente

# Modelo Computacional e a Linguagem C

**Problema:**  
Automatizar o Cálculo  
 $r = n!/d$   
e atualizar n para  
metade de r.

**Solução do Problema:**  
**Implementação em C**

## Modelo Computacional Von Neumann



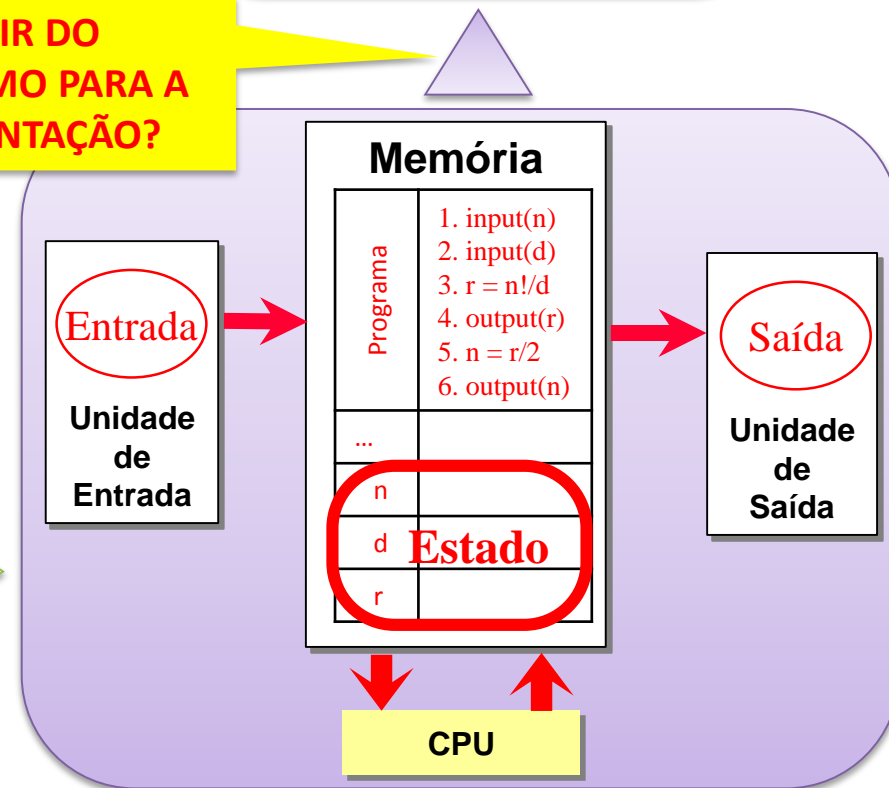
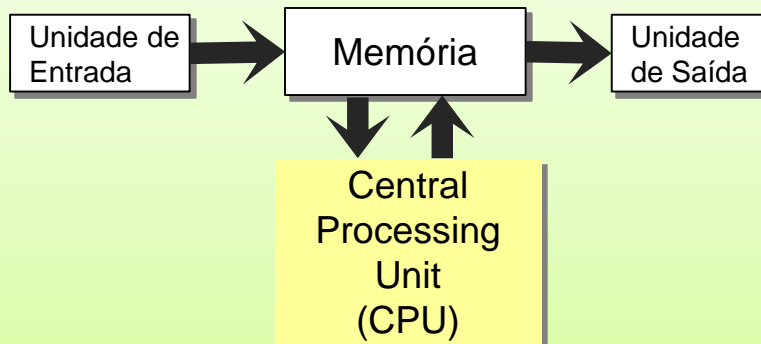
# Modelo Computacional e a Linguagem C

**Problema:**  
Automatizar o Cálculo  
 $r = n!/d$   
e atualizar n para  
metade de r.

**Solução do Problema:**  
**Implementação em C**

**COMO SAIR DO  
ALGORITMO PARA A  
IMPLEMENTAÇÃO?**

## Modelo Computacional Von Neumann

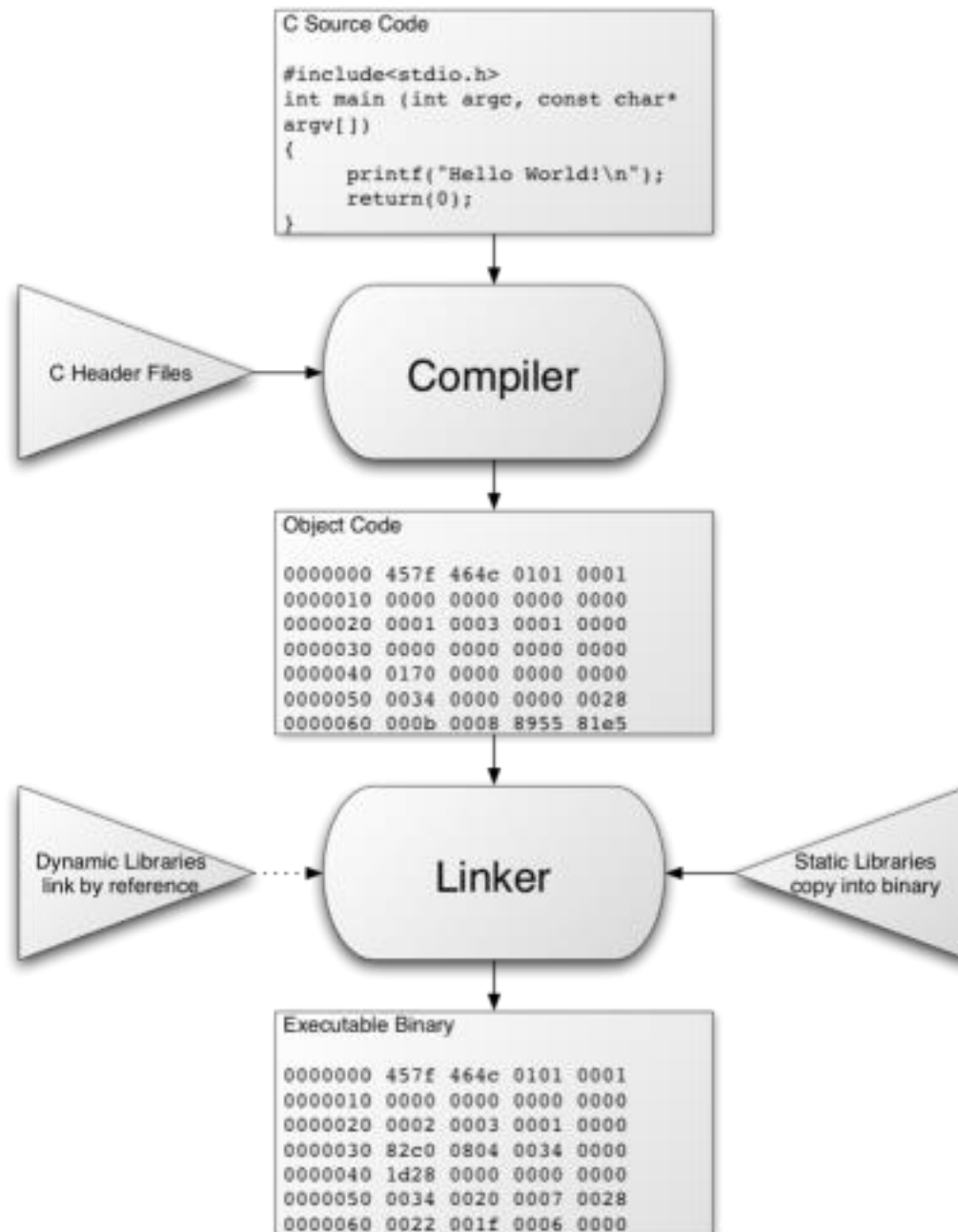


# Implementando o Programa

1. Escrever o código do programa na Linguagem C em um arquivo texto **(código fonte)**
2. Salvar o arquivo com a extensão **.c**
  - Um programa em C pode ser composto de vários códigos fontes (vários arquivos **.c**)
3. **Compilar** o código fonte para gerar o código binário/executável (arquivo com a extensão **.exe**)

## Linkedição

Para gerar o arquivo .exe é comum a geração de um código objeto (\*.obj) para cada código fonte e a posterior geração do código executável.





# Escrevendo o Programa

- Programa em C consiste de várias **funções**
  - Pode conter dezenas, centenas, ou mais, funções com **nomes únicos**
  - **NÃO** confundir o termo função com funções matemáticas!
    - Funções em C:
      - Podem executar cálculos matemáticos;
      - são conjuntos de instruções com um nome e que desempenham uma ou mais ações;
      - NÃO necessariamente devem retornar um valor.

# Exemplo

- O ponto de partida para a execução do programa é controlado pela **função** `main()`:

```
main()  
{  
}
```

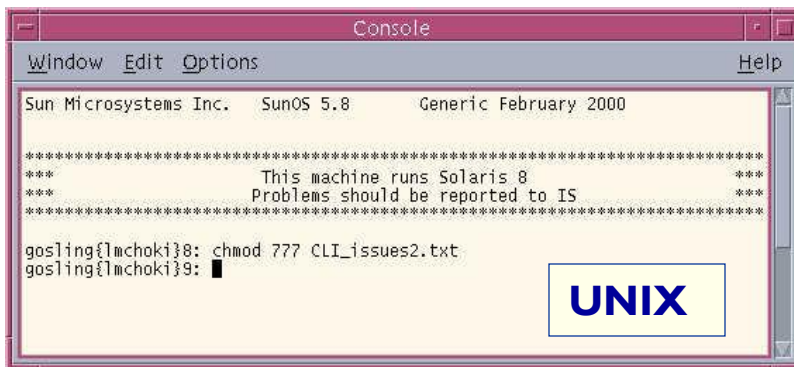
Este é o menor programa em C!

Ele INICIA e TERMINA com sucesso!

# Interfaces com o Usuário

- *User Interface (UI)*
  - *Command Line Interface (CLI)*
    - Interface de **Linha** (*Prompt*) de **Comando** (ou Shell)
    - Comum para servidores e dispositivos como roteadores
    - Usa memória de recordação
  - *Graphical User Interface (GUI)*
    - Interface **Gráfica** com o Usuário
    - Usa memória de reconhecimento (visual)

# CLI – Exemplos



```

Sun Microsystems Inc.  SunOS 5.8      Generic February 2000

*****
***                This machine runs Solaris 8                ***
***                Problems should be reported to IS          ***
*****

gosling@lmchoki:~$ chmod 777 CLI_issues2.txt
gosling@lmchoki:~$
  
```

**UNIX**



```

Microsoft(R) Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

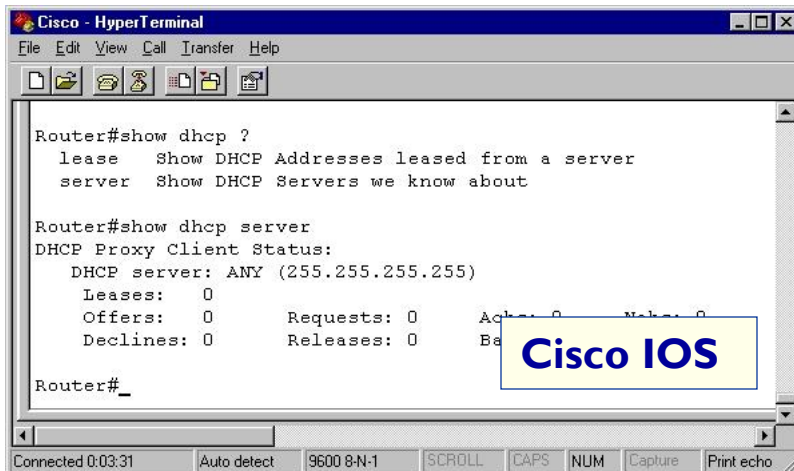
C:\WINDOWS>mk
The name specified is not recognized as an
internal or external command, operable program or batch file.

C:\WINDOWS>md
The syntax of the command is incorrect.

C:\WINDOWS>md CLI_FOLDER

C:\WINDOWS>_
  
```

**MS-DOS**



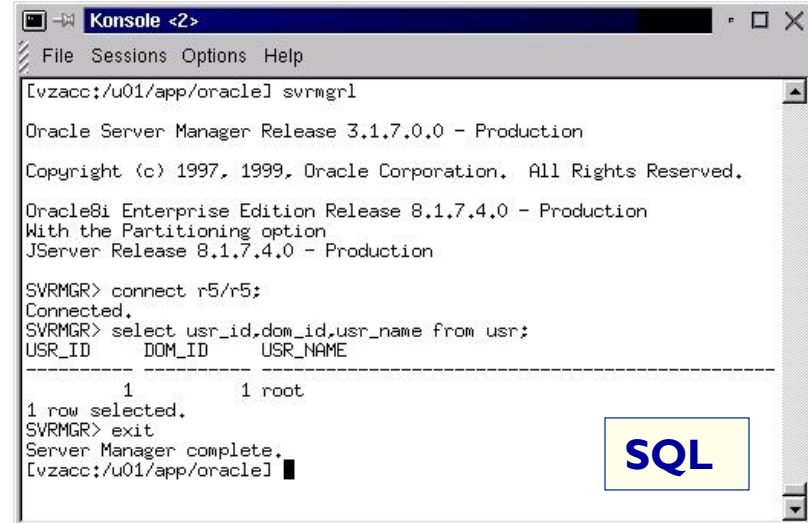
```

Router#show dhcp ?
  lease      Show DHCP Addresses leased from a server
  server     Show DHCP Servers we know about

Router#show dhcp server
DHCP Proxy Client Status:
  DHCP server: ANY (255.255.255.255)
    Leases:    0
    Offers:    0      Requests: 0      Acknowledges: 0
    Declines:  0      Releases: 0      Bounces: 0

Router#_
  
```

**Cisco IOS**



```

[vzacc:/u01/app/oracle] svrmgrl

Oracle Server Manager Release 3.1.7.0.0 - Production
Copyright (c) 1997, 1999, Oracle Corporation. All Rights Reserved.

Oracle8i Enterprise Edition Release 8.1.7.4.0 - Production
With the Partitioning option
JServer Release 8.1.7.4.0 - Production

SVRMGR> connect r5/r5;
Connected.
SVRMGR> select usr_id,dm_id,usr_name from usr;
   USR_ID   DM_ID  USR_NAME
-----
         1         1 root

1 row selected.
SVRMGR> exit
Server Manager complete.
[vzacc:/u01/app/oracle]
  
```

**SQL**

# Aplicações *Console*

- Programas que utilizam interfaces de linha de comando (CLI) são também chamados Aplicações *Console*
  - O **CONSOLE** (desde o UNIX) representa os dispositivos de **entrada** e **saída padrão** do computador, sendo:
    - Cursor de escrita (representação da **entrada**)
    - Exibição de texto (representação da **saída**)

# Outros Exemplos

```
#include <stdio.h>
main()
{
    printf("Bom Dia!");
}
```

```
#include <stdio.h>
main()
{
    int q;
    q = 3 * 3;
    printf("O quadrado de 3 eh = %d", q);
}
```

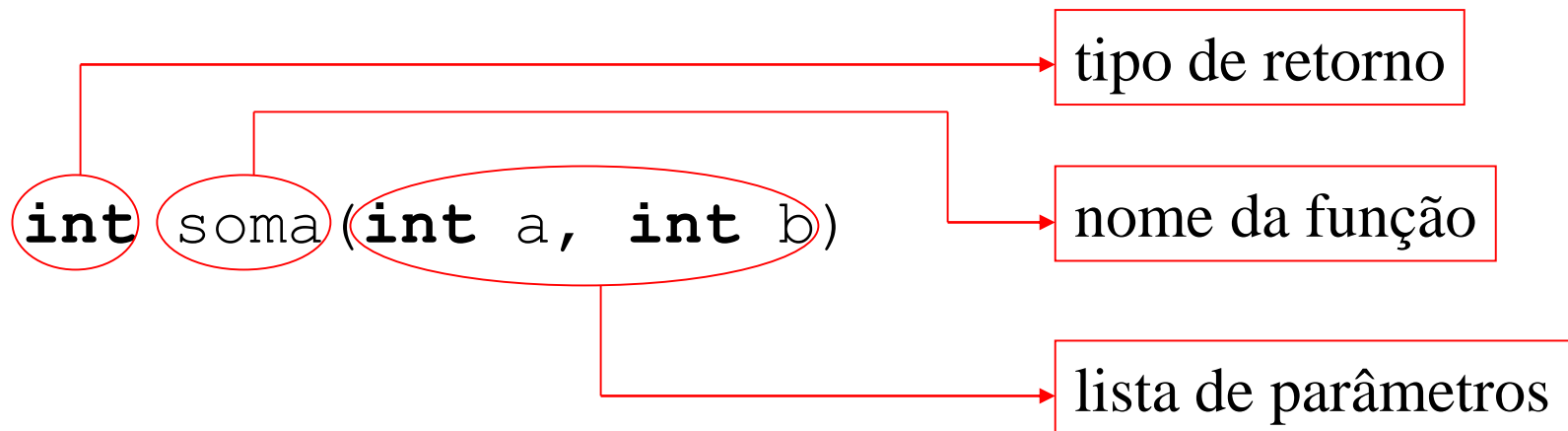
# Funções em C

- Uma função em C sempre apresenta uma assinatura e um corpo com as sentenças a serem executadas:

→ `int soma(int a, int b)`

```
{
    int c;
    c = a + b;
    return c;
}
```

# Assinatura de Funções em C





# Sentenças em C

- O **corpo** de uma função é composto de **sentenças** como: uma *declaração*, uma *instrução* ou *chamada* a uma função.
- Exemplos:

```
main() {  
    int i, j;  
    double x = 0, y = 0;  
    j = 12; i = j + 27;  
    x = soma(i,j);  
    printf("Olá!!");  
    if (i == x)  
        y = j / i;  
}
```

# Sentenças em C

- O **corpo** de uma função é composto de **sentenças** como: uma *declaração*, uma *instrução* ou *chamada* a uma função.
- Exemplos:

```
main() {  
    int i, j;  
    double x = 0, y = 0;  
    j = 12; i = j + 27;  
    x = soma(i,j);  
    printf("Olá!!");  
    if (i == x)  
        y = j / i;  
}
```

*Declarações de variáveis*

*Instruções de atribuição (muda valor de variável na memória)*

*Chamada de função*

*Instruções Condicionais*

# ATENÇÃO!

- Todo programa em C deve ter a função `main()`
  - Controla o fluxo de execução de todo o programa
    - O programa inicia com a primeira instrução de `main()` logo após a chave de abertura '{' e termina quando encontra a chave de fechamento '}'
- Um programa em C deve declarar todas as suas variáveis antes de usá-las
  - As declarações de variáveis devem vir antes das demais instruções
- Toda instrução/sentença em C é terminada por um ponto-e-vírgula

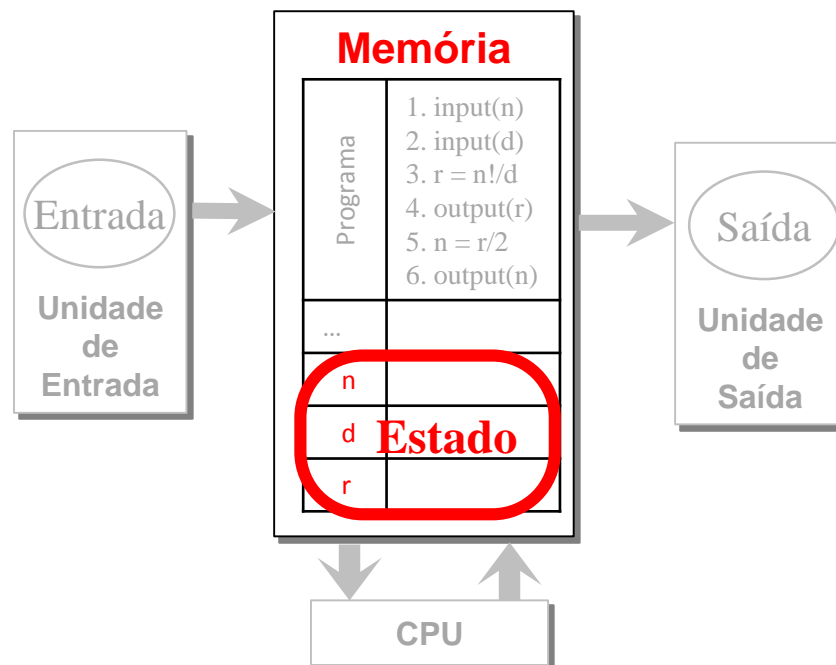
Valores Constantes

Variáveis

Identificadores

Tipos

# CONSTANTES E VARIÁVEIS



# Constantes e Variáveis

- Uma constante tem **valor** fixo e inalterável.
- Uma variável é um espaço vago, reservado e rotulado para armazenar dados (**um valor**).
  - Possui um **nome** que a **identifica unicamente**.
  - Possui um **valor** que corresponde à informação atribuída.
    - O valor de uma variável pode mudar muitas vezes durante a execução de um programa, por meio de atribuições de valor.

# Identificadores

- Identificadores são símbolos singulares utilizados nos programas.
  - O nome da função `main()` é um identificador.
  - O nome de uma variável ou de uma constante é um identificador.
  - Identificadores bem escolhidos são sua ferramenta principal para escrever programas que façam sentido.

# Identificadores

- A criação e uso de identificadores fica a cargo do programador.
  - Identificadores podem conter apenas letras minúsculas ou maiúsculas, algarismos e traço de sublinha. Além disso, identificadores não devem começar com algarismos.

# Identificadores

- Quais dos seguintes nomes são válidos para identificadores em C?

3ab

\_sim

n\_a\_o

não

char

AuToCaD

\*abc

guarda-valor

\ontem

a1

cem\*\*anos

dez^anos

A1

xFuncao

C&A



# Identificadores

- Quais dos seguintes nomes são válidos para identificadores em C?

3ab

**\_sim**

**n\_a\_o**

não

char

**AuToCaD**

\*abc

guarda-valor

\ontem

**a1**

cem\*\*anos

dez^anos

**A1**

**xFuncao**

C&A

# Identificadores

- C é sensível ao tipo de letra utilizado (distingue maiúsculas de minúsculas) em todas as letras do identificador.
  - Isso quer dizer que `minhaFuncao`, `minhafuncao` e `MinhaFuncao` são identificadores *diferentes*.
    - Por essa razão, vale a pena adotar um estilo de digitação e se manter coerente a ele.

# Palavras Reservadas

- Algumas palavras especiais da linguagem C

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			

# Tipos de Dados

- Numéricos Inteiros

---

Tipo	Tamanho	Valores
char	8 bits	-128 a +127
short	16 bits	-32.768 a +32.767
int	32 bits	-2.147.483.648 a + 2.147.483.647
long	64 bits	-9.223.372.036.854.775.808 a +9.223.372.036.854.775.807

+, -, \*, /, ^, %

# Tipos de Dados

- Numéricos de Ponto Flutuante

Tipo	Tamanho	Valores
float	32 bits	$\pm 3.40282347\text{E}+38$ $\pm 1.40239846\text{E}-45$
double	64 bits	$\pm 1.79769313486231570\text{E}+308$ $\pm 4.94065645841246544\text{E}-324$

- 3.14159, 2.0, ...
- +, -, \*, /, ...

# Tipos de Dados

- Números com e sem sinal
  - C permite que o programador defina se uma variável de tipo numérico deva ou não reservar o bit de sinal (números negativos)
  - Notação:
    - signed <tipo>
    - unsigned <tipo>
  - Se nenhum modificador for indicado, o compilador C reservará o bit de sinal

# Tipos de Dados Especiais

- Booleano (valor de predicado)
  - Pode ser utilizada qualquer variável inteira
  - O processador identificará o valor lógico/booleano como False (Falso) apenas se o valor for igual a 0
    - $0 \rightarrow \text{Falso}$
    - $\neq 0 \rightarrow \text{Verdadeiro}$
  - Variáveis booleanas são usadas principalmente como o resultado de operadores relacionais
    - $==, !=, >, <, >=, <=$

# Tipos de Dados Especiais

- Caracteres
  - Para criar variáveis que armazenam caracteres, podemos usar o tipo **char** (que possui 8 bits)
    - Internamente (na memória) será armazenado um valor inteiro representando o código de um caractere **ASCII**
  - Usando o tipo **unsigned char** que aceita até  $2^8$  valores inteiros, podemos representar 256 caracteres
    - 'a', 'b', '1', ' ', etc.



# Códigos ASCII (*ASCII Codes*)

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	<b>NUL</b> (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	<b>SOH</b> (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	<b>STX</b> (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	<b>ETX</b> (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	<b>EOT</b> (end of transmission)	36	24	044	&#36;	\$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	<b>ENQ</b> (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	<b>ACK</b> (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	<b>BEL</b> (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	<b>BS</b> (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	<b>TAB</b> (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	<b>LF</b> (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	<b>VT</b> (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	<b>FF</b> (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	<b>CR</b> (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	<b>SO</b> (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	<b>SI</b> (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	<b>DLE</b> (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	<b>DC1</b> (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	<b>DC2</b> (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	<b>DC3</b> (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	<b>DC4</b> (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	<b>NAK</b> (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	<b>SYN</b> (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	<b>ETB</b> (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	<b>CAN</b> (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	<b>EM</b> (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	<b>SUB</b> (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	<b>ESC</b> (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	<b>FS</b> (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	
29	1D	035	<b>GS</b> (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	<b>RS</b> (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	<b>US</b> (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: [www.LookupTables.com](http://www.LookupTables.com)

# Extended ASCII Codes

128	Ç	144	É	160	á	176	░	192	Ł	208	„	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	ł	209	ŧ	225	β	241	±
130	é	146	Æ	162	ó	178	▓	194	ŧ	210	π	226	Γ	242	≥
131	â	147	ô	163	ú	179		195	ı	211	„	227	π	243	≤
132	ä	148	ö	164	ñ	180	ı	196	—	212	Ł	228	Σ	244	∫
133	à	149	ò	165	Ñ	181	ı	197	+	213	ƒ	229	σ	245	∫
134	â	150	û	166	²	182	ı	198	ƒ	214	ƒ	230	μ	246	÷
135	ç	151	ù	167	°	183	π	199	ı	215	ı	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	ı	200	„	216	ı	232	Φ	248	°
137	ë	153	Ö	169	ƒ	185	ı	201	ƒ	217	ı	233	⊖	249	·
138	è	154	Ü	170	ı	186	ı	202	„	218	ı	234	Ω	250	·
139	ï	155	◊	171	½	187	ı	203	π	219	■	235	δ	251	√
140	î	156	£	172	¼	188	ı	204	ı	220	■	236	∞	252	π
141	ì	157	¥	173	ı	189	ı	205	=	221	ı	237	φ	253	²
142	Ä	158	£	174	«	190	ı	206	ı	222	ı	238	ε	254	■
143	Å	159	f	175	»	191	ı	207	ı	223	ı	239	∩	255	

Source: [www.LookupTables.com](http://www.LookupTables.com)

# Extended ASCII Codes

128	Ç	144	É	160	á	176	░	192	Ł	208	„	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	ł	209	ŧ	225	β	241	±
130	é	146	Æ	162	ó	178	▓	194	Ƨ	210	π	226	Γ	242	≥
131	â	147	ô	163	ú	179		195	└	211	„	227	π	243	≤
132	ä	148	ö	164	ñ	180	├	196	—	212	ƒ	228	Σ	244	∫
133	à	149	ò	165	Ñ	181	┤	197	+	213	ƒ	229	σ	245	∫
134	â	150	û	166	²	182	├	198	ƒ	214	π	230	μ	246	÷
135	ç	151	ù	167	°	183	π	199	├	215	†	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	┤	200	„	216	‡	232	Φ	248	°
137	ë	153	Ö	169	—	185	┤	201	ƒ	217	┘	233	⊙	249	·
						186		202	„	218	┘	234	Ω	250	·
						187	┤	203	ƒ	219	■	235	δ	251	√
						188	┘	204	├	220	■	236	∞	252	π
						189	„	205	=	221	■	237	φ	253	²
						190	┘	206	†	222	■	238	ε	254	■
						191	┘	207	└	223	■	239	∩	255	

Muitas Aplicações  
*CONSOLE* utilizam  
esses caracteres para  
“desenhar” a interface.

Source: [www.LookupTables.com](http://www.LookupTables.com)

# Tipos de Dados Especiais

- Tipo vazio (Sem Tipo)
  - Quando uma função C descreve um procedimento podemos usar o tipo **vazio** como tipo de retorno
    - indica que não possui valor de retorno
  - Representado pela palavra-chave **void**
  - Não possui valor
  - Também pode ser usado em variáveis como veremos posteriormente

# Ferramentas de Apoio

## Compiladores e Depuradores

- GCC (C Compiler)
- GCC (C++ Compiler)
- Microsoft Visual Studio Compiler
  - Microsoft Visual C++ Express Compiler
- [Intel C and C++ Compilers](#)
  - [Intel C++ Compiler for Android](#)
- [CodeWarrior](#)
- [Eclipse CDT Compiler](#)
- ...

## Editores e Ambientes

- [Code::Blocks](#)
- Microsoft Visual Studio
- DevC++ (DevCpp)
- [QtCreator](#)
- [Eclipse CDT](#)
- [NetBeans](#)
- [CodeWarrior](#)
- [CodeLite](#)
- JetBrains C++ IDE
- ...



*GNU Compiler Collection*

**GCC**

# GCC

- ***GCC (GNU Compiler Collection)***

- *Várias ferramentas para programação em várias linguagens*

- *Vários compiladores*
    - *Vários depuradores*

- **Instalação**

- <https://gcc.gnu.org/install/binaries.html>

- No **Windows**:

- Usar o **CygWin** ou o **MingW**

No nosso caso podemos deixar a instalação do MingW para o CodeBlocks (ver próximos slides)

# Instalação do GCC

- Baixar e instalar o GCC a partir do site abaixo:  
[gcc.gnu.org/install/binaries.html](http://gcc.gnu.org/install/binaries.html)
- No Linux
  - Usar gerenciador de pacotes como apt-get
- No Windows
  - Cygwin  
[sourceware.org/cygwin](http://sourceware.org/cygwin)
  - MingW  
[www.mingw.org](http://www.mingw.org)

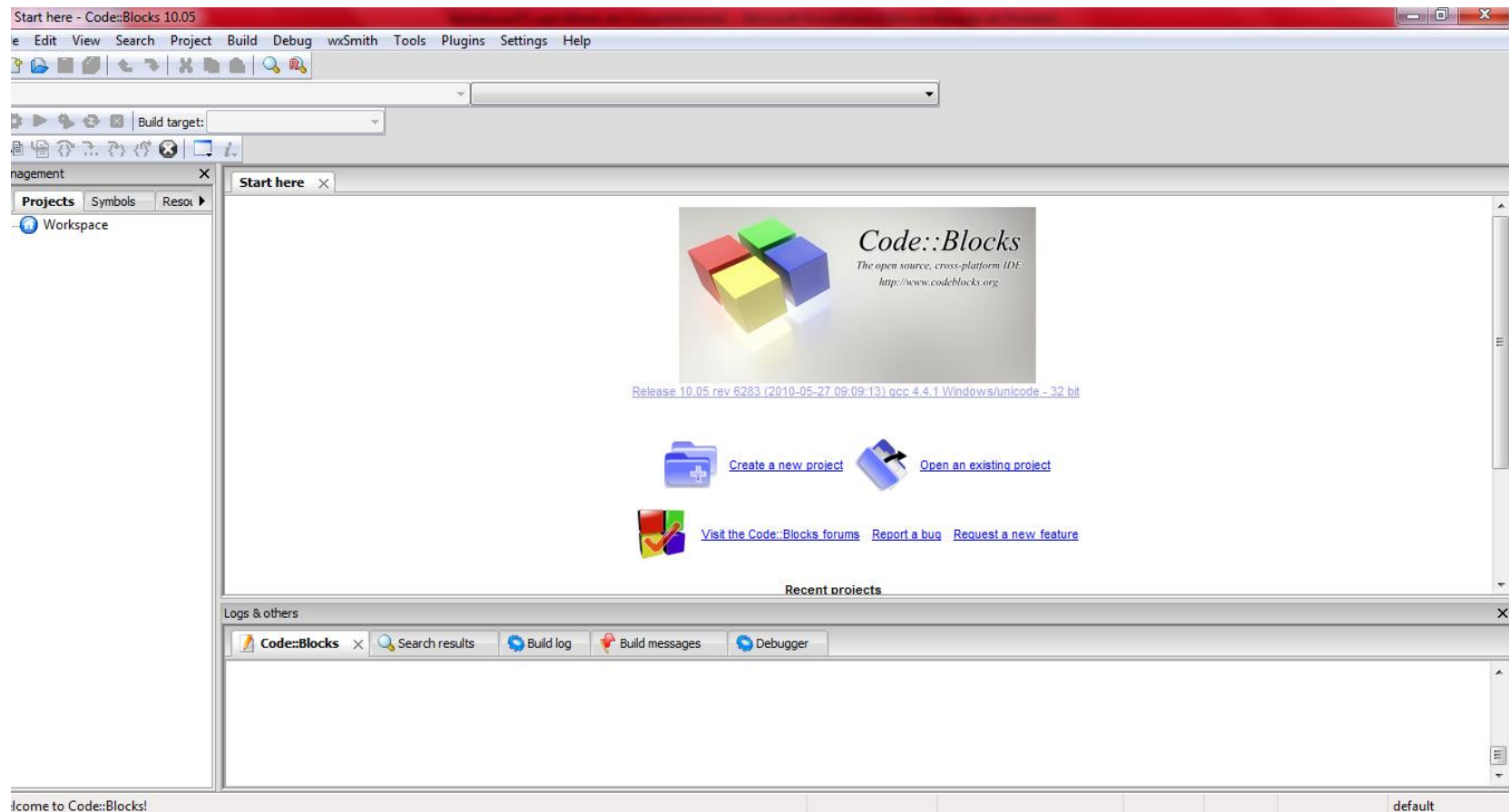
No nosso caso  
podemos deixar a  
instalação do MingW  
para o CodeBlocks  
(ver próximos slides)





# CODE::BLOCKS

# Code::Blocks IDE



# Code::Blocks IDE

- **Ambiente integrado de Desenvolvimento de Software (ADS)** para as linguagens C e C++
  - *Integrated Development Environment (IDE)*
- "Free Software"
  - Sob licença de uso da *GNU* General Public License
- Compilador
  - Usa o compilador do *GCC*

# Code::Blocks

- Links Úteis
  - [www.codeblocks.org/downloads](http://www.codeblocks.org/downloads)
  - [www.codeblocks.org/user-manual](http://www.codeblocks.org/user-manual)

# Instalação do *Code::Blocks*

- Baixar e instalar o Codeblocks no site [www.codeblocks.org/downloads](http://www.codeblocks.org/downloads)


www.codeblocks.org/downloads/26


**Main**


- Home
- Features
- Screenshots
- Downloads
  - Binaries
  - Source
  - SVN
- Plugins
- User manual
- Licensing
- Donations

**Quick links**

- FAQ
- Wiki
- Forums
- Forums (mobile)
- Nightlies
- Ticket System
- Browse SVN
- Browse SVN log



built with 



Please select a setup package depending on your platform:


- Windows 2000/XP/Vista/7/8
- Linux 32-bit
- Linux 64-bit
- Mac OS X

**NOTE:** There are also more recent *nightly builds* available in the **forums** or (for Debian and Fedora users) in **Jens' Debian repository** and **Jens' Fedora repository**. Please note that we consider nightly builds to be *stable*, usually.

**MIRRORS:** BerliOS mirrors all files *usually* at SourceForge using a "BerliOS robot" **here**. As this sometimes doesn't work, we have mirrored all file releases at SourceForge, too **here**. The latter is managed by us.

**IMPORTANT NOTE:** If you try to download from BerliOS and get a "Too many clients" - error, you should retry to download the file. According to a BerliOS - admin, this can happen several times, before the download starts. Alternatively use on of the mirrors.

**NOTE:** We have a **Changelog** for 13.12, that gives you an overview over the enhancements and fixes we have put in the new release.

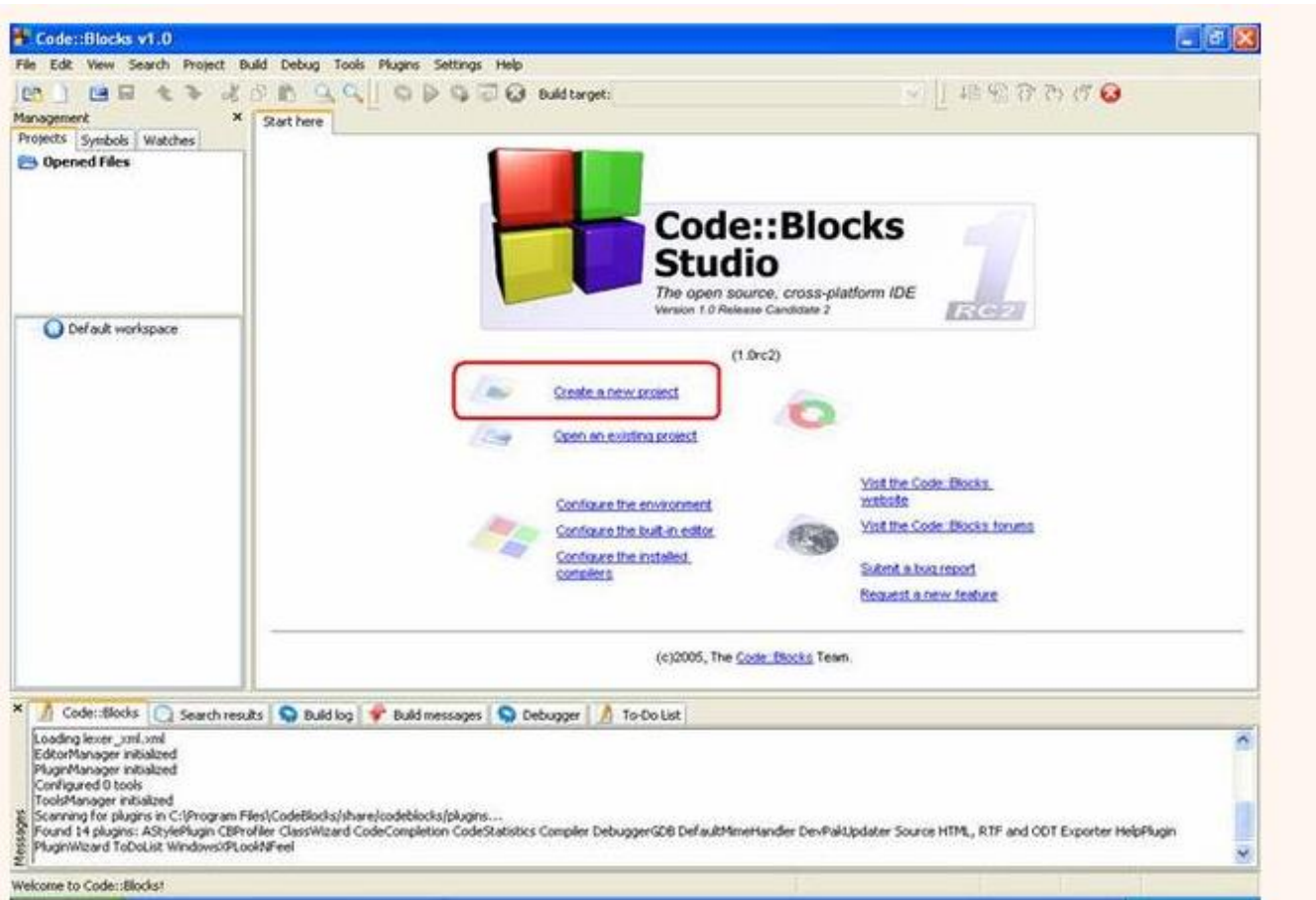
 **Windows 2000 / XP / Vista / 7 :**

File	Date	Download from
codeblocks-13.12-setup.exe	27 Dec 2013	BerliOS or Sourceforge.net
codeblocks-13.12mingw-setup.exe	27 Dec 2013	BerliOS or Sourceforge.net
codeblocks-13.12mingw-setup-TDM-GCC-481.exe	27 Dec 2013	BerliOS or Sourceforge.net

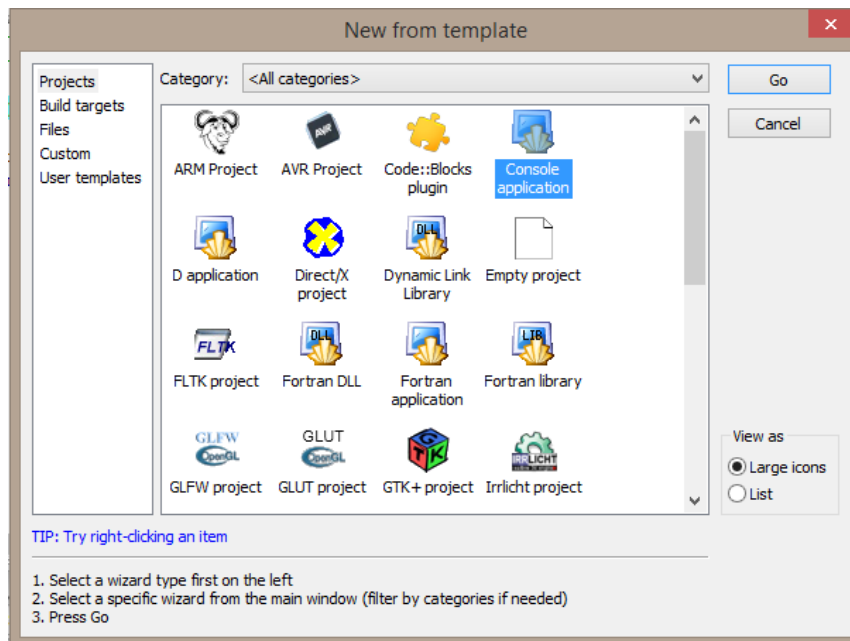
**NOTE:** The codeblocks-13.12mingw-setup.exe file *includes* the GCC compiler and GDB debugger from **TDM-GCC** (version 4.7.1, 32 bit). The codeblocks-13.12mingw-setup-TDM-GCC-481.exe file includes the TDM-GCC compiler, version 4.8.1, 32 bit. While v4.7.1 is rock-solid (we use it to compile C::B), v4.8.1 is provided for convenience, there are some known bugs with this version related to the compilation of Code::Blocks itself.

**IF UNSURE, USE "codeblocks-13.12mingw-setup.exe"**

# Criando Projetos

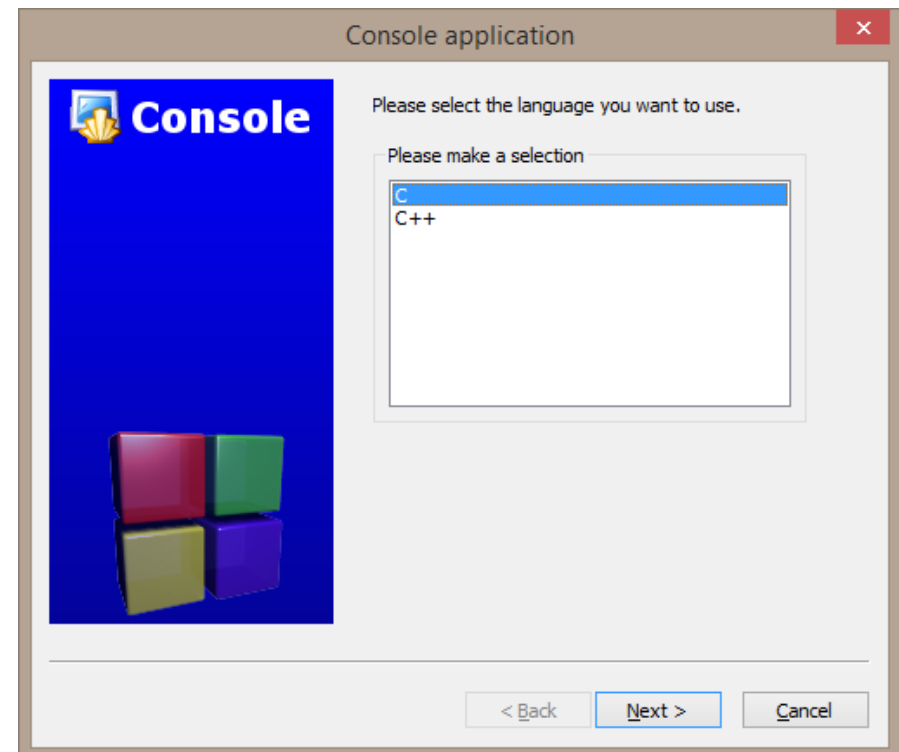


# Projeto de Aplicação *Console*

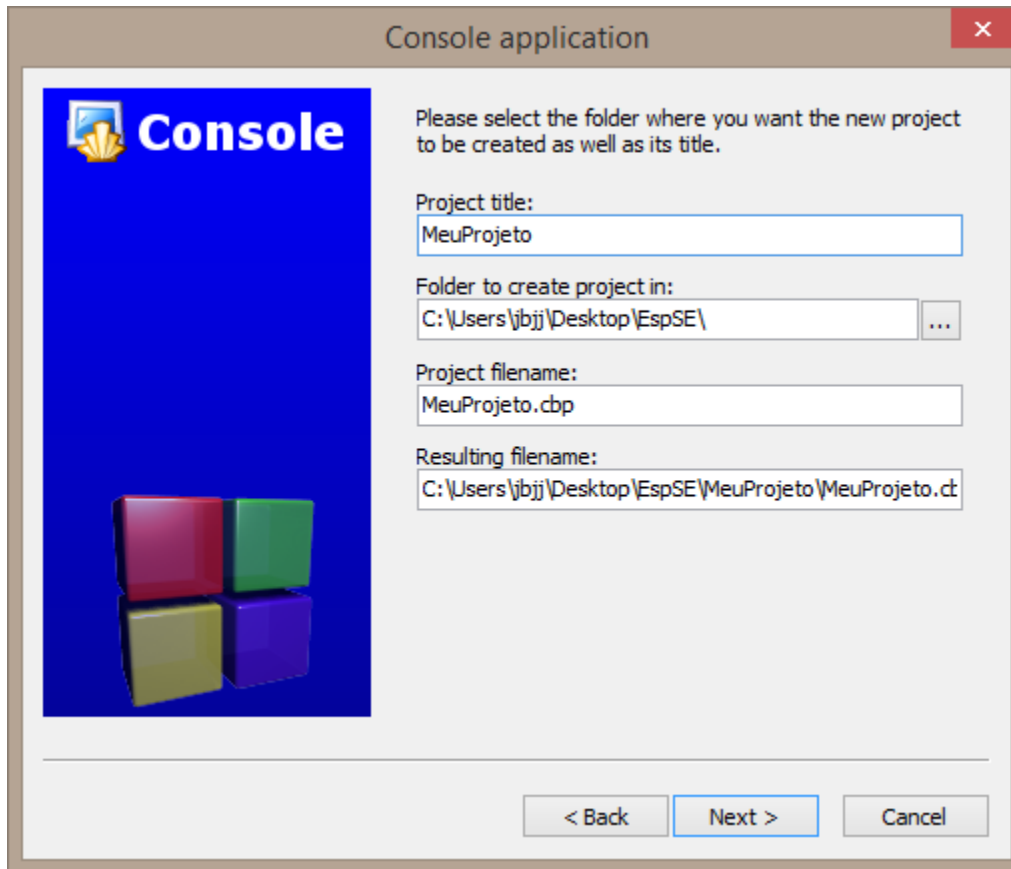


1. Escolha do tipo de Projeto.

2. Escolha da Linguagem:  
No nosso caso, C



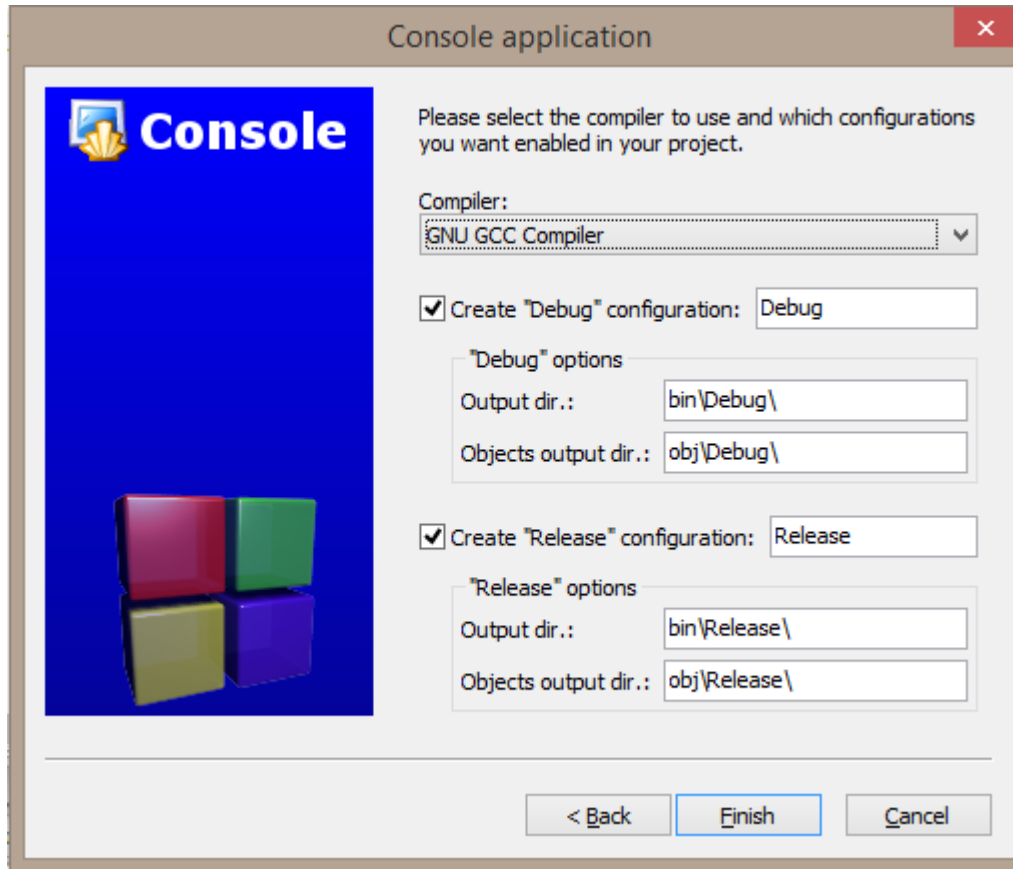
# Definição do Projeto



- No Code::Blocks, um projeto é representado por um arquivo com a extensão **.cbp**
- Na pasta indicada para salvar o projeto será criada uma pasta com o nome do projeto, a qual possuirá o arquivo do projeto.



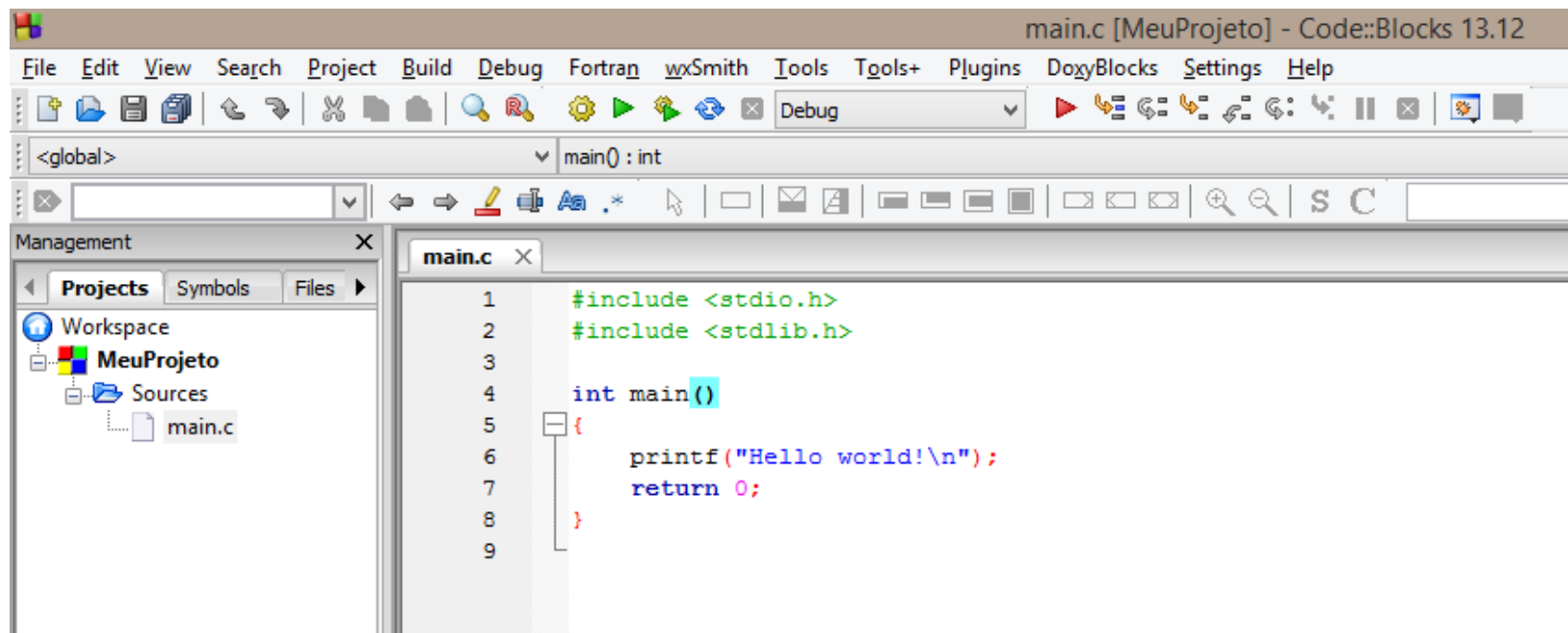
# Escolha do Compilador



- No Code::Blocks, o compilador padrão é o GCC.
  - É possível escolher outro compilador
- É possível definir as opções de compilação:
  - **Debug** – Permite **Depuração** (Execução Passo-a-Passo do código)
  - **Release** – Geração do binário final para **Entrega** ao cliente)

# main.c

- Ao criar o projeto, o Code::Blocks cria um arquivo de código fonte (***source file***) main.c



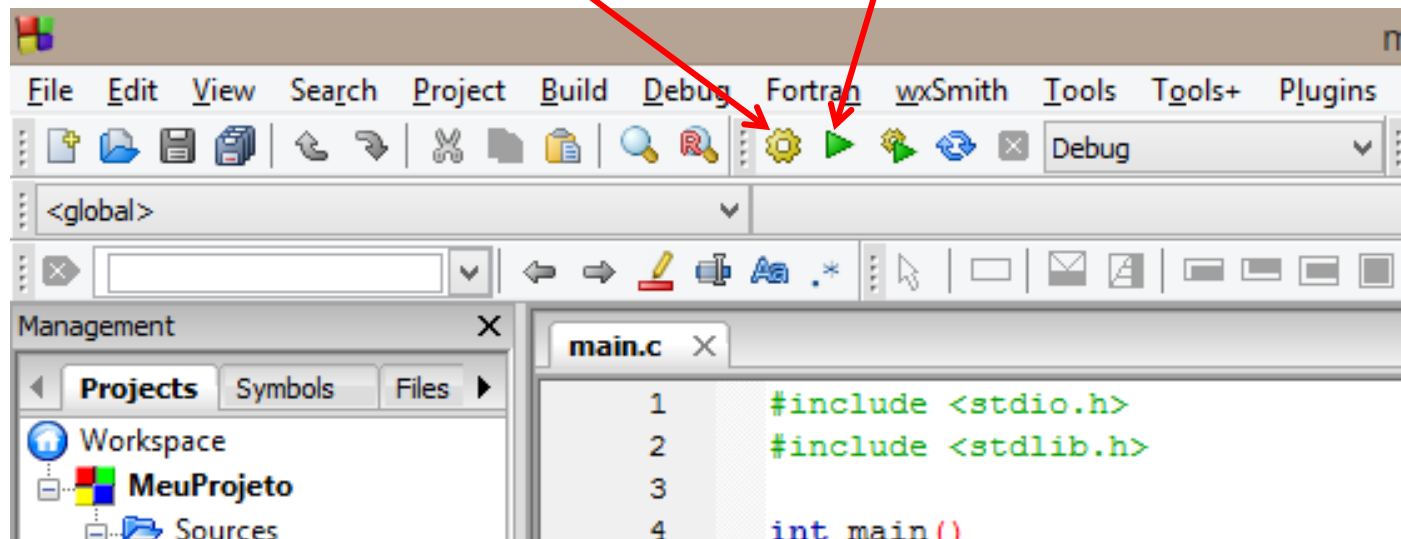
# Escrevendo código no CodeBlocks

1. Crie um arquivo novo utilizando o menu *File* → *New* → *Empty file*
2. Escreva/Digite o código lembrando sempre de endentá-lo adequadamente (usando a tecla **TAB**) e comentá-lo apropriadamente. Note que o Code::Blocks tende a endentar o código automaticamente, mas haverá vários momentos em que a endentação devesse ser corrigida manualmente
3. Salve o arquivo após digitar o cabeçalho, utilizando um nome apropriado terminado com a extensão *.c* . Para tanto, use o menu *File* → *Save File* (**CTRL + S**)

# Compilando o Programa

***Build*** (Compilar)

***Run*** (Executar)



# Atalhos

- ***Build and Run***



- Use a tecla **F9** (ou o menu *Build* → *Build and Run*) para compilar e executar o programa.

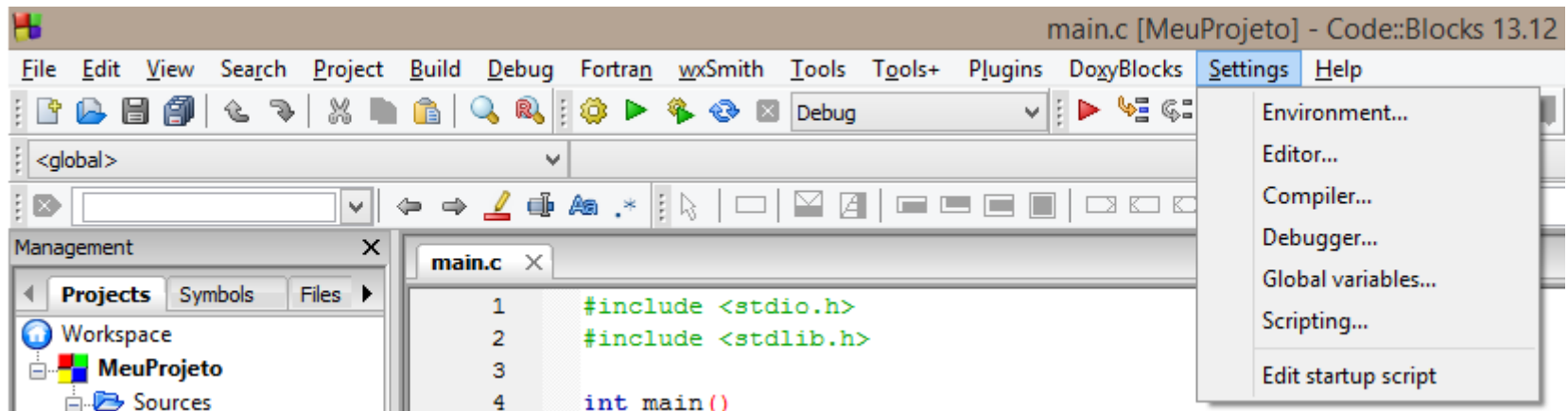
- Exibir/Ocultar ***Logs***

- Para exibir ou ocultar os ***Logs*** (registros de informações sobre a compilação), pressione a tecla **F2**.

# Configurações

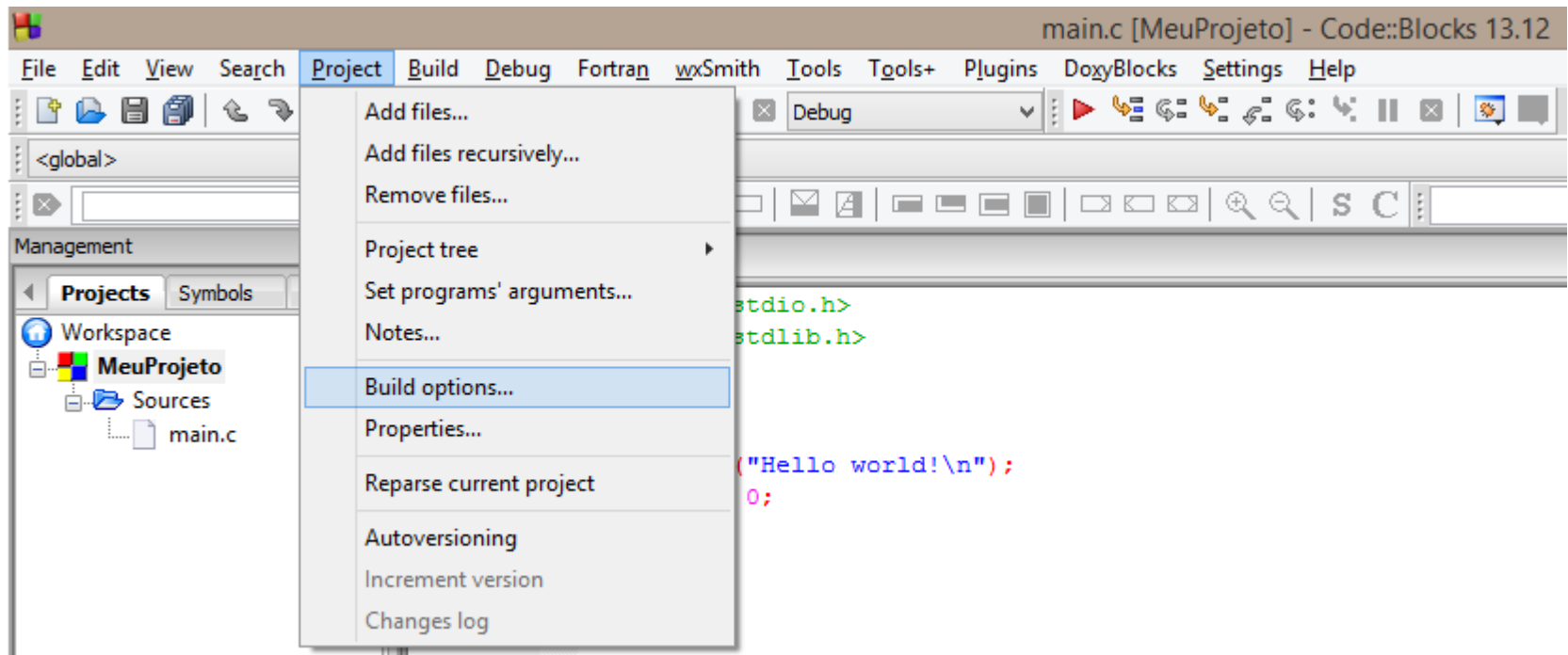
## Globais do Code::Blocks

- No Menu Settings é possível alterar o compilador (Settings → Compiler) ou o depurador (Settings → Debugger) que seu Code::Blocks poderá utilizar



# Configurações do Projeto Code::Blocks

- No menu *Project* → *Build options* podemos alterar as configurações do projeto atual



# Exercício

1. Fazer um programa (aplicativo ***console***) que imprime na tela o texto **"Bom dia!"**
2. Compilar com o GCC usando a linha de comando (o ***prompt*** de comando ou ***shell***)
  - No CygWin digite:  
gcc cygdrive/c/<diretório>/<nomeFonte>.c  
-o cygdrive/c/<diretório>/<nomeBinário>.exe
3. Executar o Programa
  - No Cygwin digite: ./cygdrive/c/<diretório>/<nomeBinário>



# Exercício

1. Criar um projeto no Code::Blocks para o código fonte existente do exercício anterior
2. Adicionar o arquivo arquivo do exercício anterior ao projeto ou substituir o **main.c** pelo seu arquivo anterior.
3. Compilar o projeto usando o Code::Blocks
4. Executar o projeto usando o Code::Blocks

# Exercício

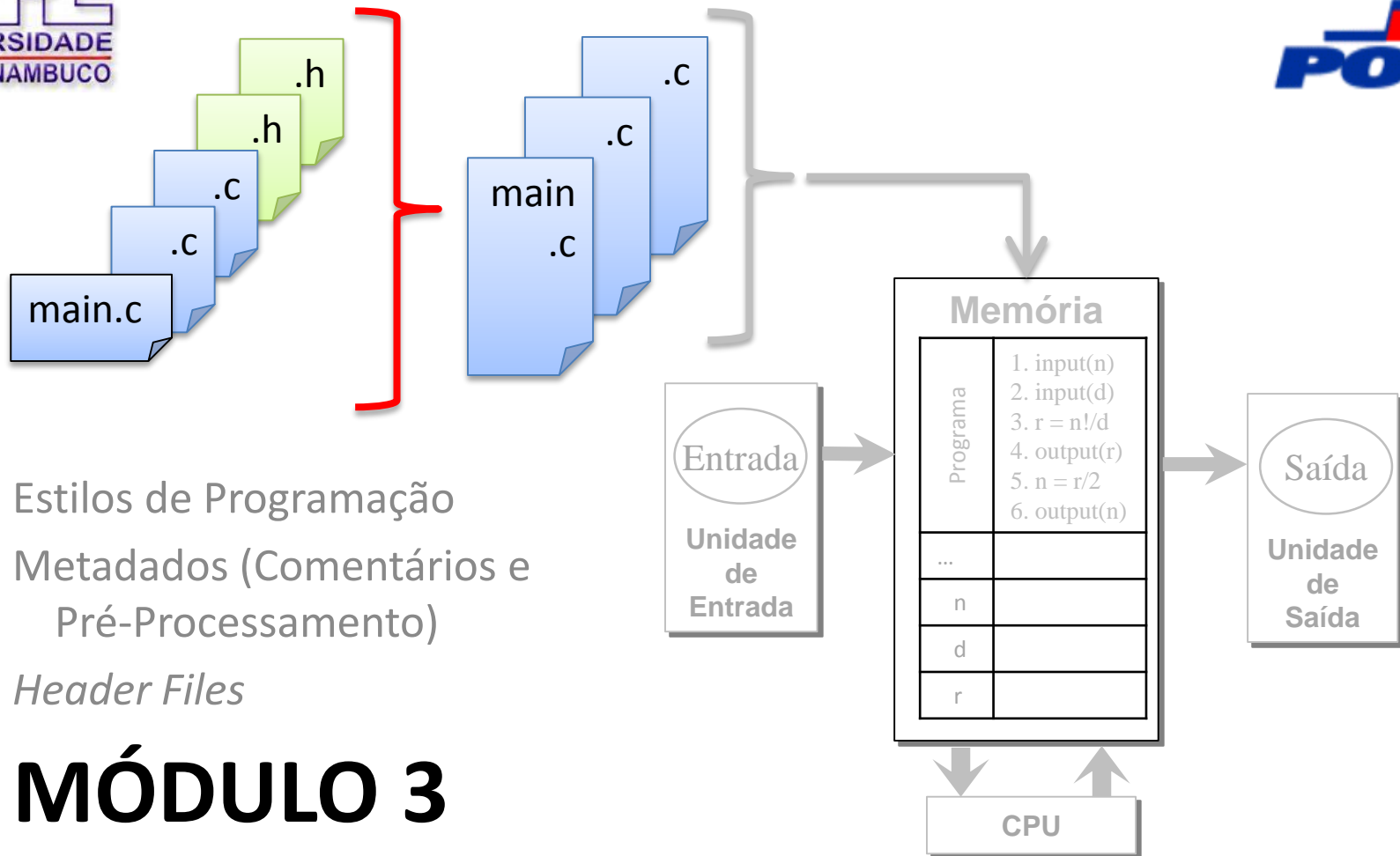
1. Escreva o programa abaixo que utiliza Constantes e Variáveis

```
main()  
{  
    const double PI = 3.1415;  
    double raio, area;  
    raio = 4.0;  
    area = PI * raio * raio;  
}
```

2. Compilar e ver os resultados.

# Exercício

1. Altere o programa anterior para tentar atribuir novo valor à constante PI após o cálculo da área.
2. Compilar e ver os resultados.



# Estilos de Programação

- Ao escrever seu programa, você pode colocar espaços, tabulação e pular linhas à vontade, pois o compilador ignora estes caracteres.
  - Em C não há um estilo obrigatório
- Na disciplina, teremos
  - Guia de estilo em C

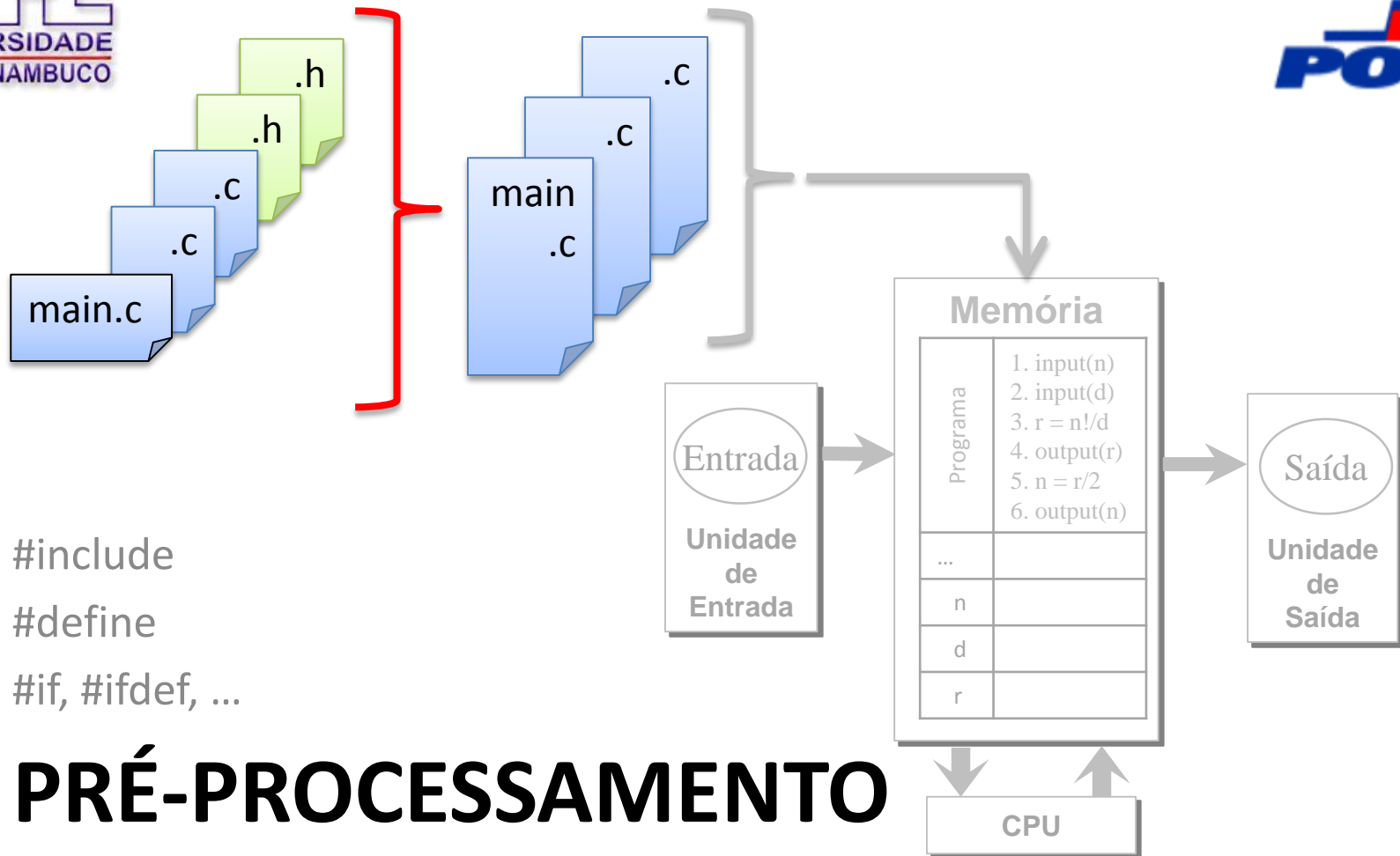
# Comentários

- É sempre bom adicionar comentários para explicar pontos de um programa
  - Ignorados pelo compilador
  - Importante para os programadores
  - Exemplo:

```
main()  
{  
    /*comentário sobre os próximos passos */  
    ...  
    //comentários de uma linha  
}
```

# Exemplos

```
main()  
{  
    /* Comentário sobre a constante PI  
       descrevendo ...  
    */  
    const double PI = 3.1415;  
    /*const*/ double sqrt2 = 1.41;  
    double raio, area /*, volume */;  
    raio = 4.0; // Comentário no fim da linha  
    area = PI * raio * raio;  
    // TODO volume = area * altura;  
    // Comentários de linha única isolado  
    ...  
}
```



#include  
#define  
#if, #ifdef, ...



# Diretivas de pré-processamento

## #include

- A diretiva #include permite incluir (importar) funções definidas em outros arquivos (normalmente, *Header Files*)

```
#include <stdio.h>
#include "minhasFuncoes.h"

main ()
{
    ...
}
```

# Diretivas de pré-processamento

## #define

- A diretiva  
#define  
permite  
definir

### ***macros***

(regras de  
substituição  
de texto)

```
#define UM 1  
#define DOIS UM+UM
```

```
main()  
{
```

```
...
```

```
y = DOIS;
```

```
x = y + UM;
```

```
z = x + UM;
```

```
...
```

```
}
```

```
main()  
{
```

```
...
```

```
y = 2;
```

```
x = y + 1;
```

```
z = x + 1;
```

```
...
```

```
}
```

# Diretivas Condicionais

`#if`

- A diretiva `#if` permite enviar um trecho de código para o compilador se uma condição for verdadeira

`#else`

- A diretiva `#else` permite tratar o caso falso da condição definida na diretiva `#if` anterior

`#elif`

- Equivale a um `#else #if`

`#endif`

- Toda diretiva condicional deve ser delimitada pela diretiva `#endif` que finaliza o bloco de verificação

# Diretivas Condicionais de Definição

## #ifdef

- A diretiva #ifdef permite verificar se uma macro foi definida anteriormente (no arquivo atual ou num arquivo previamente incluso)

## #ifndef

- A diretiva #ifndef permite verificar se uma macro NÃO foi definida anteriormente

# Exemplo

```
#ifdef DEBUG
```

```
    print("Depurando Versão 1");
```

```
#else
```

```
    print("Versão 1 (em Produção)");
```

```
#endif
```

# Outras Diretivas de pré-processamento

#undef

#line

<https://gcc.gnu.org/onlinedocs/cpp/>

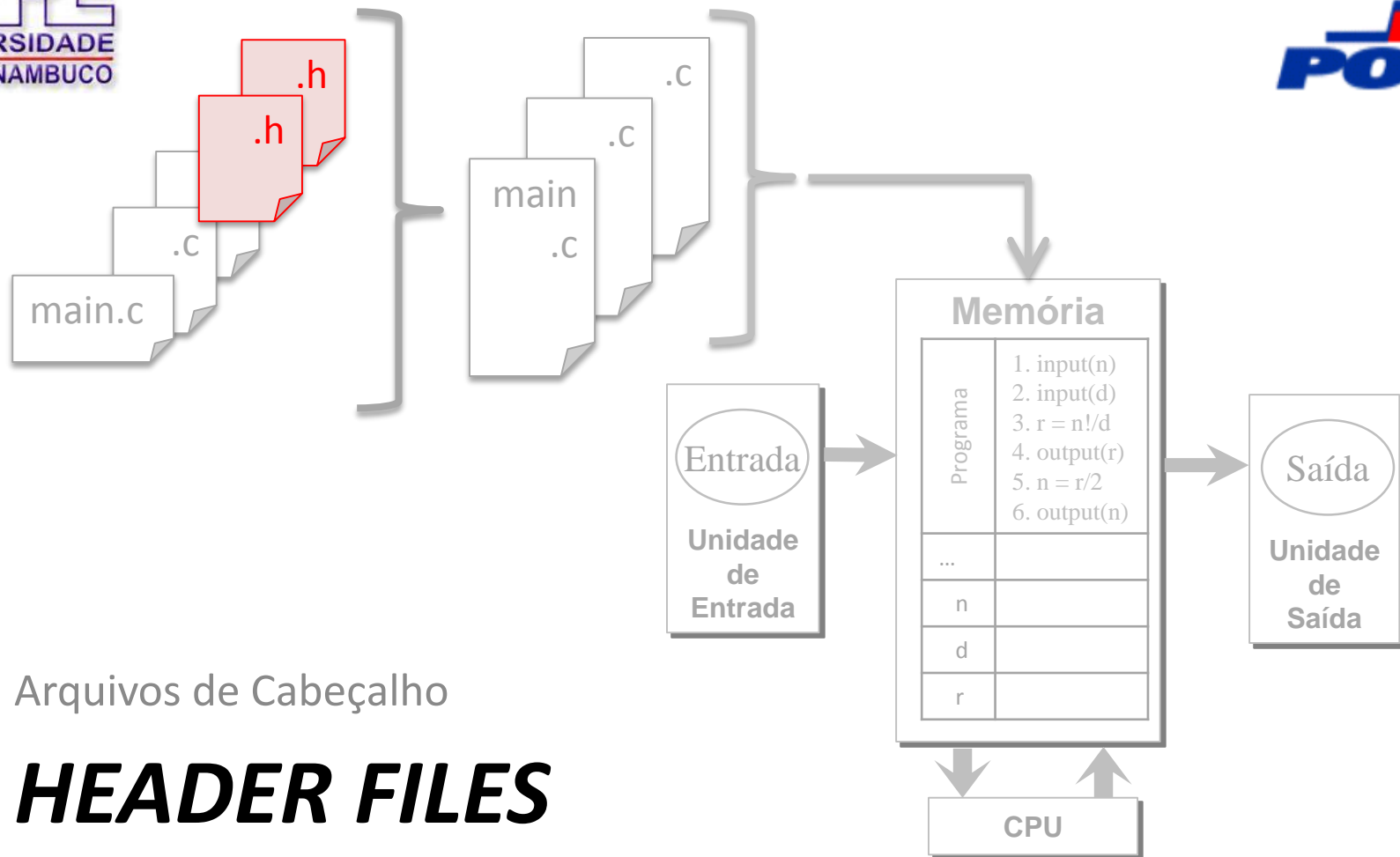
#error

#pragma

- Diretiva definida pelo compilador específico
- C99 definiu um padrão mínimo: #pragma STDC ...

---

#include\_next    #warning    #import



Arquivos de Cabeçalho

# HEADER FILES

# Header Files

- **Header Files** permitem estruturar/organizar Grandes Projetos
  - **Dividir para Conquistar**
    - Arquivos menores facilitam manutenção e compilação
  - Agrupamento
    - Cada arquivo fonte deve ter uma coleção de funções relacionadas e coerentes
    - Exemplo: imagens.h, audio.h, graficos.h, ...



# Header Files

- Com exceção do arquivo main.c outros códigos fonte devem ser separados em:

## *<meuArquivo>.h*

- Possuirá tipos definido pelo programador (como *typedef's*, *enum's* e *struct's*)
- Possuirá assinaturas/protótipos de funções (function prototypes) para cada função cujo corpo está definido em *<meuArquivo>.c*

## *<meuArquivo>.c*

- Possuirá as definições (corpo) de funções
- Pode possuir assinaturas de funções usadas apenas no próprio arquivo *<meuArquivo>.c*

# ATENÇÃO

1. Salve os **header files** (.h) a mesma pasta (**folder**) que estão seus arquivos fonte (.c)
  - Use `<...>` para incluir um **header file** definido pela biblioteca padrão C
    - Exemplo: `#include <stdio.h>`
  - Use `"..."` para incluir um **header file** próprio
    - Exemplo: `#include "minhasFuncoes.h"`

# Header File – Estrutura

```
#ifndef CODE_NAME
#define CODE_NAME
... typedef's ...
... structs ...
... enums ...
... prototypes ...
...
#endif
```

## ATENÇÃO:

observe o uso das diretivas condicionais para garantir que esse **Header File** só será usado (copiado e colado antes da compilação) uma vez.

# Relembrando a Definição de Funções

- Uma função em C possui assinatura e um corpo

```
int soma(int a, int b)
```

```
{
    int r;
    r = a + b;
    return r;
}
```

```
main()
```

```
{
    int r = soma(3, 4);
}
```

# Definindo Funções

```
float soma(float a, float b);  
  
void main()  
{ printf("A soma de 3 com 4 = %f\n", soma(3,4) );  
}  
  
float soma(float a, float b)  
{ float s;  
  s = a + b;  
  return s;  
}
```

programa.c

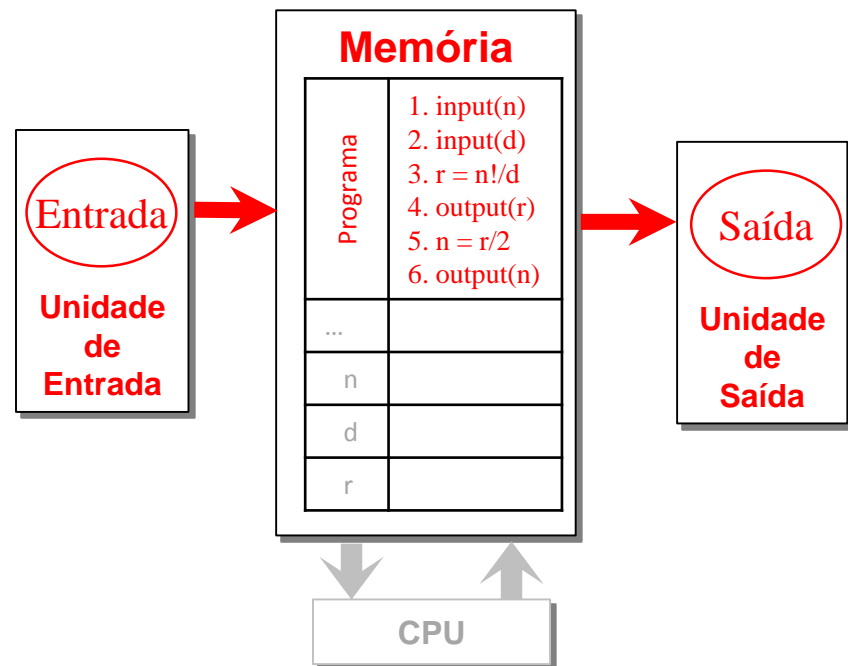
Saída

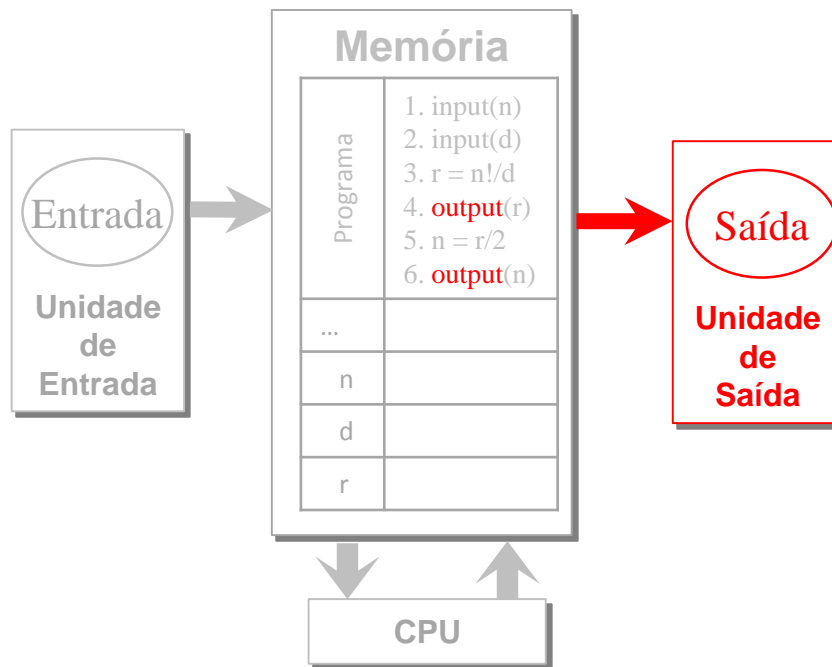
Entrada

Instruções Básicas

- Operadores Aritméticos
- Operadores de Atribuição
- Operadores Relacionais
- Operadores Lógicos
- ...

## MÓDULO 4





Funções C para Saída

# SAÍDA DE DADOS

# A Função printf()

- Utilizada para enviar sequências de caracteres para a saída padrão de um programa em C
- Sintaxe:
  - `printf("string de controle", lista de argumentos);`



# A Função printf()

- Exemplos:

```
printf("Uma linha");
```

```
printf("Uma linha\nDuas linhas");
```

```
printf("Os números são: %d e %d\n", 7, 8);
```

```
printf("%s está a %d Km de Recife", "Caruaru",  
120);
```

```
printf("%d%% de %d = %f\n", p, x, x*(p/100.0));
```

# A Função printf()

- Caracteres especiais:

<code>\n</code>	avanço de linha
<code>\t</code>	tabulação (tab)
<code>\b</code>	retrocesso (backspace)
<code>\"</code>	aspas duplas
<code>\\</code>	barra

# A Função printf()

- Códigos de impressão formatada:

<code>%c</code>	caractere
<code>%d</code>	inteiro
<code>%e</code>	notação científica
<code>%f</code>	ponto-flutuante
<code>%o</code>	octal
<code>%x</code>	hexadecimal
<code>%u</code>	inteiro sem sinal
<code>%s</code>	string
<code>%%</code>	o caractere '%'

# A Função printf()

- O tamanho de campos de impressão é indicado logo após o '%' e antes do tipo do campo:

```
printf("Os alunos são %4d\n", 44);  
printf("Os alunos são %04d\n", 44);  
printf("Os alunos são %-4d\n", 44);  
printf("R$ %.2f\n", 1234.5632);  
printf("R$ %10.2f\n", 1234.5632);  
printf("R$ %-10.2f\n", 1234.5632);
```

# Exercício

- Escreva um programa C que
  - declare 3 variáveis caractere e atribua a elas as letras a, b e c.
  - Declare também 3 variáveis inteiras e atribua os valores 1, 2 e 3 a elas.
  - Declare uma variável de ponto flutuante para guardar a média aritmética das 3 variáveis inteiras declaradas anteriormente.
  - Imprimir na tela todas as variaveis.

# Exercício

- Escreva um programa que imprima na tela:

um

dois

três

quatro

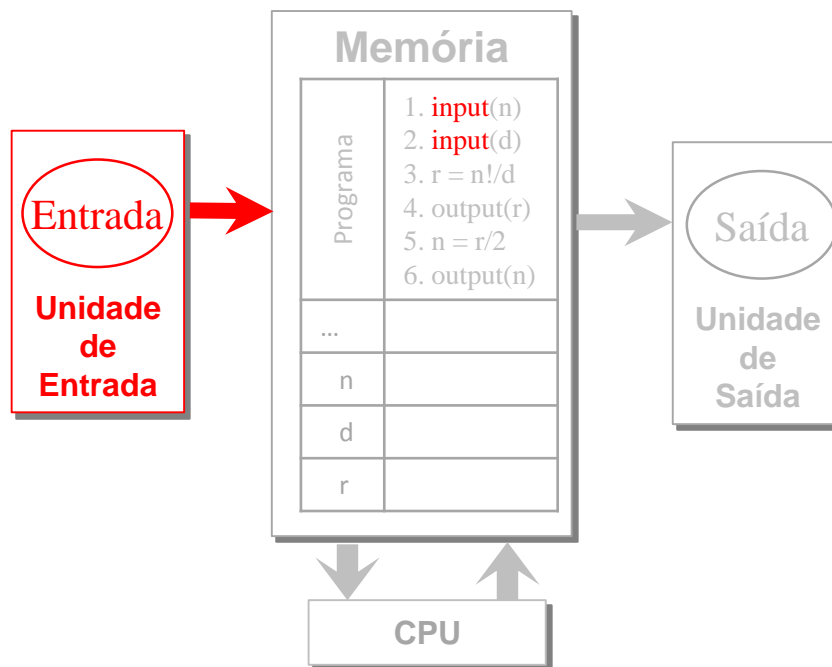
# A função `putchar()`

- A função `putchar()` imprime um caractere na tela
- Ao invés de usar:

`printf("%c", c)`

Usa-se:

`putchar(c)`



Funções C para Entrada

# ENTRADA DE DADOS



# A Função scanf()

- Semelhante à função printf(), exceto que é utilizada para leitura de dados
  - Formatação na direção contrária

- Sintaxe:

`scanf("string de controle", endereço dos argumentos);`

- Exemplo:

```
int anos;
```

```
printf("Digite sua idade em anos: ");
```

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

 endereço da variável

# Endereços de Variáveis

- Um endereço de memória é visto como um número inteiro sem sinal
- O código para formatação de um endereço é %u
- Exemplo:

```
int num = 2;
```

```
printf("Valor=%d, endereço=%u", num, &num);
```

# Outro exemplo

- Exemplo:

```
int dia, mes, ano;
```

```
printf("Entre com a data: ");
```

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

```
printf("A data foi: %d/%d/%d\n",dia,mes,ano);
```

# Exercício

- Escreva um programa que solicite a idade de uma pessoa e imprima na tela quantos dias aproximadamente esta pessoa já viveu.
  - OBS.: Considere que todos os anos possuem 365 dias.

# As funções getche() e getch()

- Em algumas situações, a função scanf() não se adapta perfeitamente pois é preciso pressionar <enter> depois da entrada
- As funções getche() e getch() efetuam a leitura de um caractere e continuam a execução do programa
  - Biblioteca conio.h
  - getche() apresenta o caractere lido na tela, enquanto que getch() não apresenta

```
char c1 = getch();  
char c2 = getche();
```

Pode ser útil no  
final do  
programa.

# Exercício

- Escreva um programa que peça para o usuário digitar um caractere na tela, imprima o caractere digitado na mesma linha e, por fim, imprima em linhas diferentes:
  - seu valor na tabela ASCII
  - seu antecessor e o valor dele na tabela ASCII
  - seu sucessor e o valor dele na tabela ASCII

Use as funções `getche()` e `getch()` – duas opções

# A função `getchar()`

- A função `getchar()` lê o primeiro caractere de um string e termina quando a tecla <enter> for pressionada
- Ao invés de usar:

`scanf("%c", &c)`

Usa-se:

`c = getchar()`

# Exercício

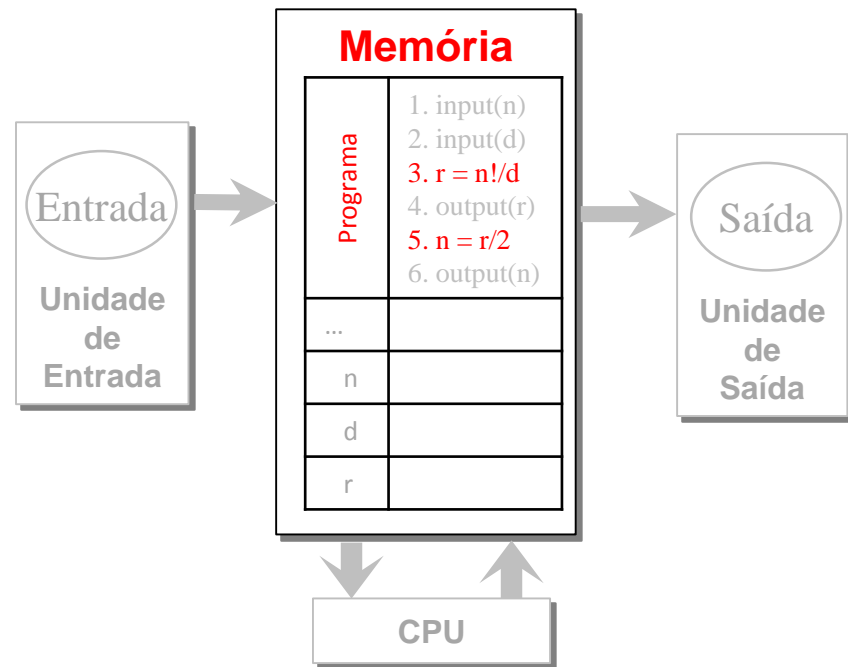
- Escreva um programa que peça ao usuário para escrever um string na tela e que escreva o primeiro caractere do string na linha seguinte



Operadores Aritméticos  
Operadores de Atribuição  
Operadores Relacionais  
Operadores Lógicos

...

# INSTRUÇÕES BÁSICAS



# Operadores Aritméticos

- Operadores Binários

= atribuição

+ adição

- subtração

\* multiplicação

/ divisão

% resto da divisão (módulo)

- Operador Unário

- menos unário

# Operador de Atribuição

- O operador de atribuição não tem equivalente na matemática
- Exemplo: `num = 2000;`
  - atribui o valor 2000 à variável num
  - `2000 = num;` não faz sentido em C!
- C aceita várias atribuições em uma mesma instrução
  - `a = b = c = 10;`

# Exercício

- Escreva um programa que solicite ao usuário uma temperatura em graus Fahrenheit e imprima o equivalente em graus Celsius

$$\text{Celsius} = (\text{Fahrenheit} - 32) * 5 / 9$$

# Incremento e Decremento

- **++** *soma 1 ao seu operando*
- **--** *subtrai 1 do seu operando*

- **Exemplo:**

```
int num1, num2;  
num1 = 5;  
num2 = ++num1;  
printf("num1=%d,num2=%d", num1, num2);
```

- O resultado será  
num1=6, num2=6

# Incremento e Decremento

- **ATENÇÃO:** Se o operador for pós-fixado, o resultado será diferente!
- Exemplo:

```
int num1, num2;  
num1 = 5;  
num2 = num1++;  
printf("num1=%d,num2=%d", num1, num2);
```
- O resultado será:  
num1=6, num2=5

# Exemplo 1

- Qual a execução do programa a seguir?

```
void main()  
{ int n;  
  printf("Digite um número inteiro: ");  
  scanf("%d", &n);  
  printf("Os números são: %d %d %d\n", n, n+1, n++);  
  printf("N= %d\n", n);  
}
```

> programa

Digite um número inteiro: 5

Os números são: 5 6 5

N= 6

## Exemplo 2

- Teste agora

```
void main()  
{ int n;  
  printf("Digite um número inteiro: ");  
  scanf("%d", &n);  
  printf("Os números são: %d %d %d\n", n, n+1, ++n);  
  printf("N= %d\n", n);  
}
```

> programa

Digite um número inteiro: 5

Os números são: 6 7 6

N= 6



# Operadores Aritméticos de Atribuição

<code>num += 2</code>	equivale a	<code>num = num + 2</code>
<code>num -= 2</code>	equivale a	<code>num = num - 2</code>
<code>num *= 2</code>	equivale a	<code>num = num * 2</code>
<code>num /= 2</code>	equivale a	<code>num = num / 2</code>
<code>num %= 2</code>	equivale a	<code>num = num % 2</code>

# Operadores Aritméticos de Atribuição

$x *= y + 1$       *equivale a*  $x = x * (y+1)$

$t /= 2.5$       *equivale a*  $t = t/2.5$

$p \%= 5$       *equivale a*  $p = p\%5$

Qual será o valor de x, y e z?

```
int x=1, y=2, z=3;
```

```
x += y += z += 7;
```

# Precedência

- Alguns operadores tem uma prioridade maior de execução que outros.
- Qual será o valor de cada variável abaixo? (assuma que todas são do tipo int)

$x = (2+1)*6;$

$y = (5+1)/2*3;$

$i = j = (2+3)/4;$

$a = 3+2*(b=7/2);$

$c = 5+10\%4/2;$

# Coerção de Tipos (*Type Casting*)

- O uso de variáveis de tipos diferentes em C **pode causar problemas!**

– Exemplo:

```
main( )  
{  
    int x,y;  
    float z;  
  
    x=2;  
    y=3;  
    z=(x+y)/2;  
    printf ("z = %f\n", z);  
}
```

# Coerção de Tipos (*Type Casting*)

- **Solução:** usar Coerção (**Cast**)

– Exemplo:

```
main( )  
{  
    int x,y;  
    float z;  
  
    x=2;  
    y=3;  
    z=(float)(x+y)/2;  
    printf ("z = %f\n", z);  
}
```

# Operadores Relacionais

- Retornam um valor booleano
  - > maior
  - >= maior ou igual
  - < menor
  - <= menor ou igual
  - == igual
  - != diferente

# Operadores Lógicos

- Operam sobre valores booleanos

&&                      E             $(0 \ \&\& \ 1 \ == \ 0)$

||                      OU         $(0 \ || \ 1 \ == \ 1)$

!                      NÃO     $(!0 \ == \ 1)$    */\* (1 == 1) \*/*

- Exemplos:

```
int a = 2, b = 5;
```

```
int v1 = (a > 0) && (b != a);
```

```
int v2 = !v1;
```

```
int v3 = !(a < 0);   /* (a >= 0) */
```

# Operadores Lógicos

## Bit-a-Bit

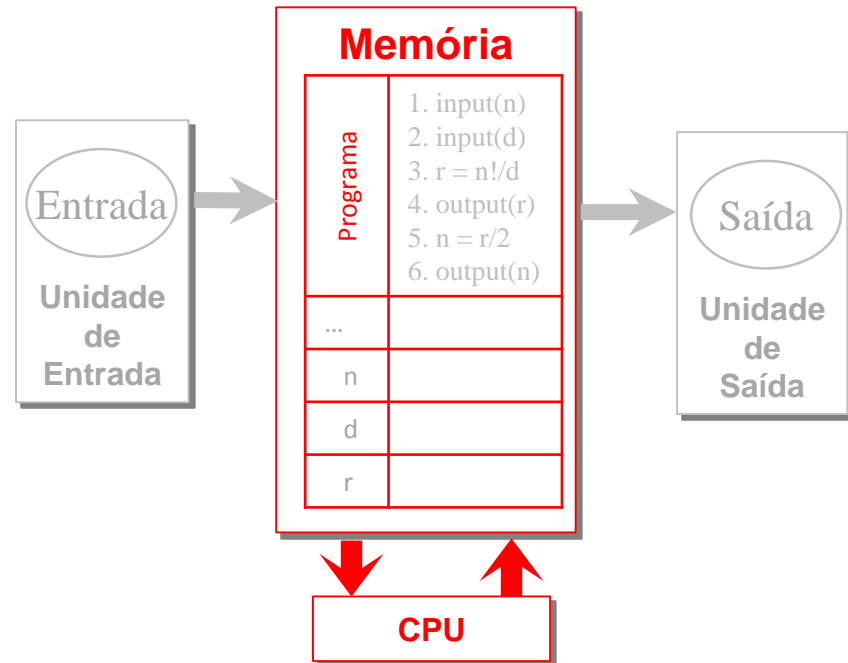
Operador	Ação
&	AND
	OR
^	XOR (OR exclusivo)
~	NOT
>>	Deslocamento de bits à direita
<<	Deslocamento de bits à esquerda



Instruções de Controle de Fluxo

- Tomada de Decisões
- Repetições

## MÓDULO 5

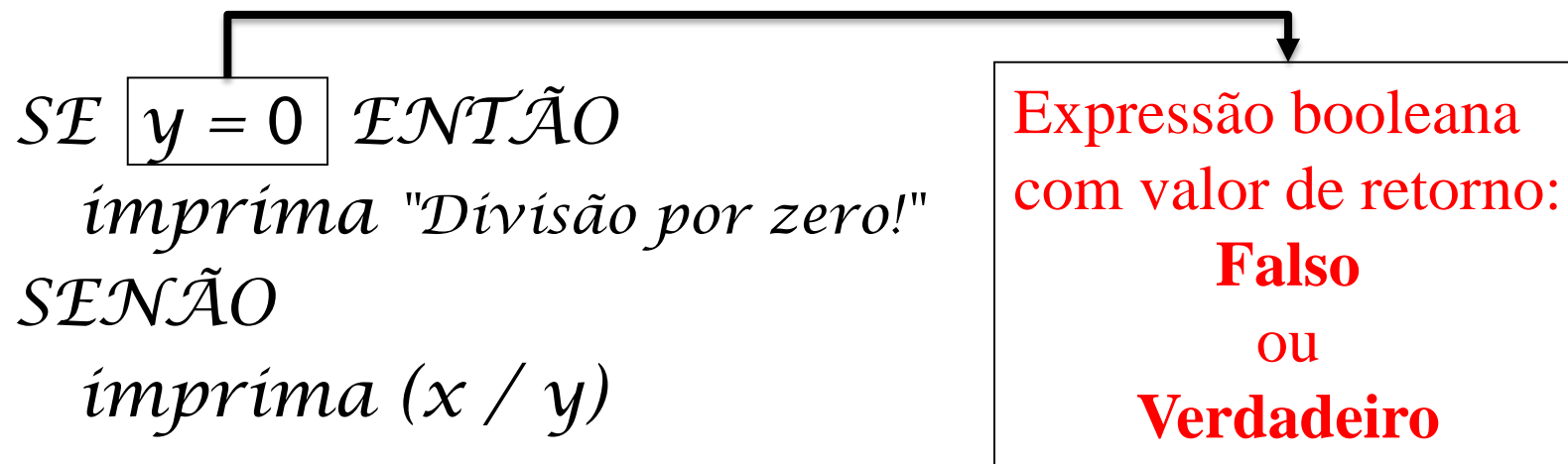


# Instruções de Controle

- Toda linguagem de programação precisa oferecer pelo menos três formas básicas de controle:
  1. Executar uma sequência de instruções
    1. Exemplo: declarações, definições, atribuições, chamadas de funções, ...
  2. Tomada de Decisões - realizar testes para decidir entre ações alternativas
  3. Repetições - repetir uma sequência de instruções

# Tomada de Decisões

- Permite decidir entre ações alternativas usando testes que envolvem o uso de **operadores lógicos e relacionais**, guiando o fluxo de execução de um programa



# Tomada de Decisões

- Permite decidir entre ações alternativas usando testes que envolvem o uso de **operadores lógicos e relacionais**, guiando o fluxo de execução de um programa

*SE  $y = 0$  ENTÃO*

*imprima "Divisão por zero!"*

Alternativa a ser executada se  $y = 0$  for **Verdadeiro**

*SENÃO*

*imprima  $(x / y)$*

Alternativa a ser executada se  $y \neq 0$  for **Falso**

# Comandos de Decisão

- C oferece **4** instruções de decisão:
  - `if`
  - `if-else`
  - `switch`
  - operador condicional (**operador ternário**)  
`<condição> ? <expressão 1> : <expressão 2>`

# Comando `if`

- **`if`**

`if` (expressão de teste)  
comando ou bloco

- **`if-else`**

`if(expressão_de_teste)`  
    *comando ou bloco 1*  
`else`  
    *comando ou bloco 2*

# Exemplo

```
if (y == 0) {  
    printf("Divisão por zero!");  
} else {  
    printf(x / y);  
}
```

```
if (y == 0)  
    printf("Divisão por zero!");  
else  
    printf(x / y);
```

Se o bloco tiver apenas um comando, podemos omitir as chaves.

# Exemplo

```
if (a >= b) {  
    c = a - b;  
    printf("a é maior ou igual a b");  
} else {  
    c = b - a;  
    printf("b é maior que a");  
}
```



# Exemplo

```
char ch = getche();  
if (ch == 'p')  
{  
    printf("\n Você digitou a tecla 'p'");  
    printf("\n Digite qualquer tecla ");  
    printf("para terminar...");  
    getch();  
}
```

## Exemplo 2

```
char ch = getche();  
if (ch == 'p')  
    printf("\n Você digitou a tecla 'p'");  
else  
    printf("\n Você digitou a tecla '%c'",ch);  
printf("\n Digite qualquer tecla ");  
printf("para terminar...");  
getch();
```

# Exercício

- Elabore um algoritmo que dada a idade de um nadador (lida do teclado) imprime a categoria na qual ele está:
  - infantil A = 5 - 7 anos
  - infantil B = 8-10 anos
  - juvenil A = 11-13 anos
  - juvenil B = 14-17 anos
  - adulto = maiores de 18 anos

# Exercício

- Desenvolva um programa para implementar uma calculadora com quatro operações utilizando comandos `if-else` para identificar qual das quatro operações deve ser realizada.

# Comando `switch`

- O comando `if-else` é útil para a escolha de uma entre duas alternativas
- Quando mais de duas alternativas são necessárias, pode ficar deselegante utilizar vários `if-else` encadeados
  - Para estes casos o comando **`switch`** é a melhor opção

# Comando switch

- switch

```
switch(expressão_constante)  
{  
    case constante1:  
        comando ou bloco 1  
        break;  
    ...  
    default:  
        comando ou bloco  
}
```

☒ int, char, short e long  
☐ float e double

# Exemplo

```
int codigo;
printf("Digite o código da operação: ");
scanf("%d", &codigo);
switch(codigo)
{
    case 1 : printf("Extrato de Conta Corrente");
             tira_extrato();
             break;
    case 2 : printf("Transferência");
             transfere_dinheiro();
             break;
    default: printf("Código inválido");
}
}
```

## Cuidado com o **break**!

```
switch (nota)
{ case 'A': printf("Excelente\n");
  case 'B' : printf("Bom\n");
  case 'C' : printf("OK\n");
}
```

- O código acima imprime todas as mensagens:

Excelente

Bom

OK



# Exercício

- Como você faria uma calculadora usando o comando `switch`?

# Solução

```
float total, a, b;
char operador;
printf("Digite: número operador número\n");
scanf("%f %c %f", &a, &operador, &b);
switch(operador)
{
    case '+': total = a + b; break;
    case '-': total = a - b; break;
    case '*':
    case 'x': total = a * b; break;
    case '/': total = a / b; break;
    default: printf("Operador desconhecido\n");
             total = 0.0;
}
printf("Total da operação = %f\n", total);
```

# Operador Condicional

- É uma expressão (**resulta em um valor**) no formato

*<condição> ? <expressão 1> : <expressão 2>*

- A <expressão 1> será avaliada caso <condição> seja verdadeira. Caso contrário, <expressão 2> será avaliada.

- Exemplo:

```
int a = 17 + 15, b = 3 * 7;
```

```
unsigned short maior = (a > b) ? a : b;
```

# Exercício

- O que será impresso pelo programa a seguir?

```
void main()  
{ int num = -42;  
  printf("O valor é %d\n",  
        (num>0) ? 0 : ++num );  
  printf("O valor é %d\n", num);  
}
```

# Exercício

- Faça um programa que receba 3 números e imprima-os na ordem correta
  - Pode usar qualquer comando de decisão/seleção
- Exemplo:
  - Se o usuário entrar com : 9 10 3
    - Saida será: "3 < 9 < 10"

# Repetições

- Laços são utilizados para repetir uma sequência de instruções.
- Exemplo:

*ENQUANTO* houver refrigerantes *FAÇA*

pergunte qual refrigerante o cliente deseja

receba o dinheiro

forneça o refrigerante

devolva o troco

# Instruções de Repetição

- A linguagem C oferece 3 tipos de laços:
  - **for**
  - while
  - do-while
    - todos eles fazem a mesma coisa, ou seja, executa uma mesma sequência de instruções sempre que uma condição for satisfeita

# Instruções de Repetição

- O laço for engloba 3 expressões
  - inicialização
    - executada uma única vez no início do laço
  - teste
    - condição que controla o laço; o laço será executado enquanto esta condição for verdadeira
  - incremento
    - define como a variável de controle do laço será alterada



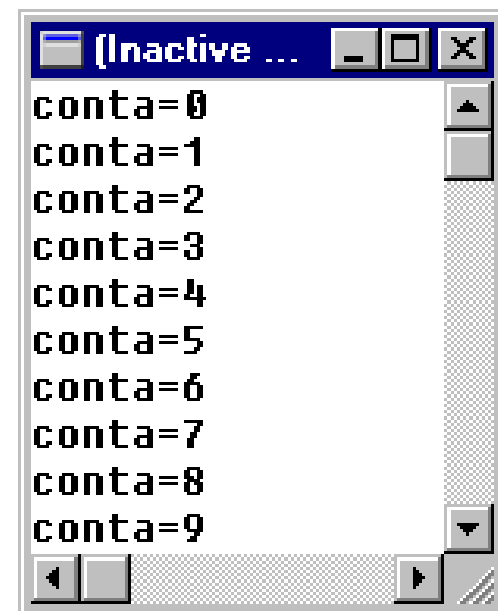
# Instruções de Repetição

- O laço *for*

`for(inicialização; teste; incremento)`  
*comando ou bloco*

```
for(conta=0; conta<10; conta++)  
    printf("conta=%d\n", conta);
```

instrução a ser repetida



```
(Inactive ...  
conta=0  
conta=1  
conta=2  
conta=3  
conta=4  
conta=5  
conta=6  
conta=7  
conta=8  
conta=9
```

# Exercício

- Faça um programa que imprima em ordem decrescente todos os valores inteiros maiores que zero a partir de um número fornecido pelo usuário

## Exercício (Resposta)

```
void main() {  
    int c, n;  
    printf("Forneça um inteiro positivo: ");  
    scanf("%d",&n);  
    if(n >= 0) {  
        for(c = n; c > 0; c--)  
            printf("%d ",c);  
    }  
    else  
        printf("Valor negativo.\n");  
}
```

# Exemplo

inicialização      ponto-e-vírgula      teste      incremento      não coloque ponto-e-vírgula aqui

```
int conta, total;
for(conta = 0, total = 0; conta < 10; conta++)
{
    total += conta;
    printf("conta = %d, total = %d\n", conta, total);
}
```

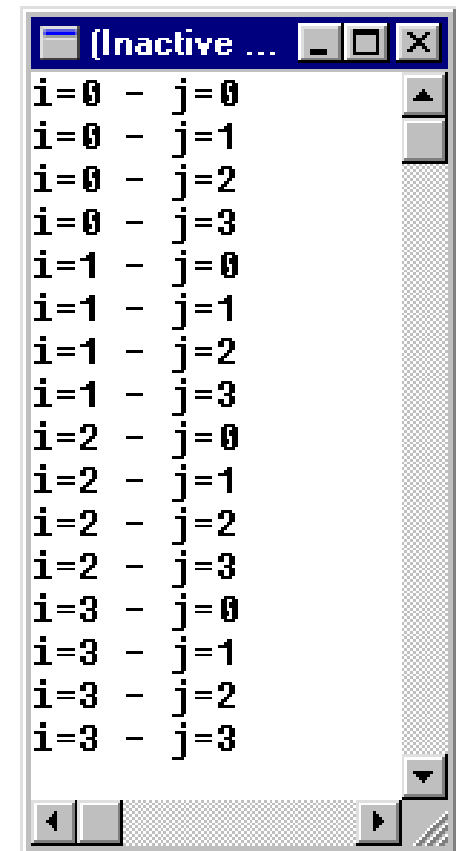
corpo do laço

```
conta = 0, total = 0
conta = 1, total = 1
conta = 2, total = 3
conta = 3, total = 6
conta = 4, total = 10
...
```

# Laços Aninhados

- Quando um laço está dentro do escopo de outro, diz-se que o laço interior está aninhado

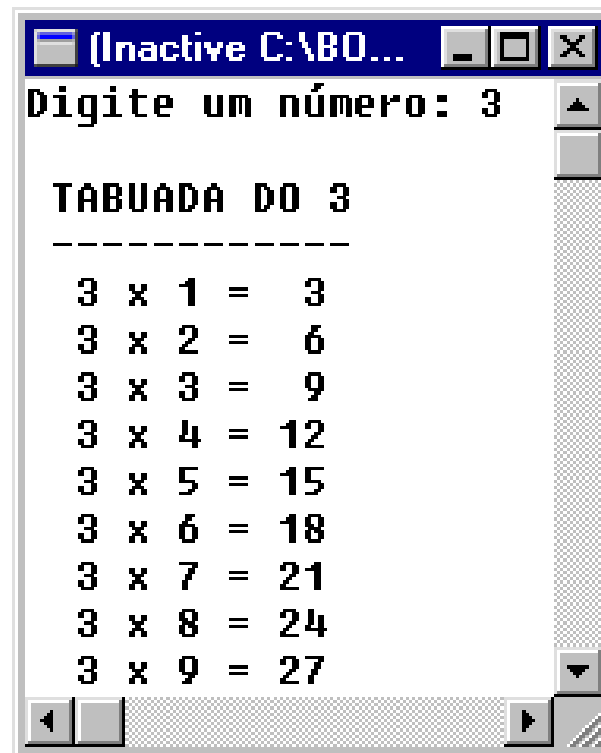
```
for(i = 0; i < 4; i++)  
    for(j = 0; j < 4; j++)  
        printf("i=%d - j=%d\n",i,j);
```



```
i=0 - j=0  
i=0 - j=1  
i=0 - j=2  
i=0 - j=3  
i=1 - j=0  
i=1 - j=1  
i=1 - j=2  
i=1 - j=3  
i=2 - j=0  
i=2 - j=1  
i=2 - j=2  
i=2 - j=3  
i=3 - j=0  
i=3 - j=1  
i=3 - j=2  
i=3 - j=3
```

# Exercício 1

- Faça um programa para imprimir a tabuada de um número fornecido pelo usuário



```
(Inactive C:\BO...
Digite um número: 3

TABUADA DO 3
-----
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
```

## Exercício 2

- Modifique o programa anterior para solicitar do usuário dois números. O programa deverá imprimir as tabuadas de todos os números compreendidos no intervalo dado pelo usuário. Por exemplo, se o usuário entrar com 4 e 7, o programa deverá imprimir a tabuada do 4, depois a tabuada do 5, a seguir a do 6 e por último a tabuada do 7.

## Exercício 3

- Modifique o programa anterior para imprimir as tabuadas dos números em forma de coluna, ou seja, uma ao lado da outra.



# Instruções de Repetição

- A linguagem C oferece 3 tipos de laços:
  - for
  - **while**
  - **do-while**
    - todos eles fazem a mesma coisa, ou seja, executa uma mesma sequência de instruções sempre que uma condição for satisfeita

# O comando/laço *while*

`while (expressão_de_teste)`  
*comando ou bloco*

```
int c, total=0;
c = 0;
while(c < 10)
{ printf("c é igual a %d\n",c);
  c+=17;
  total++;
}
printf("O laço foi executado %d vezes.\n",total);
```

# while Versus for

```
for (conta=0; conta<10; conta++)  
    printf("conta=%d\n", conta);
```

```
conta=0;  
while (conta<10)  
{  
    printf("conta=%d\n", conta);  
    conta++;  
}
```

**Equivalentes**

## while Versus for

- O laço `for` é mais indicado quando o número de iterações for conhecido antecipadamente
- O laço `while` é mais apropriado quando a iteração possa ser terminada inesperadamente, em consequência das operações do corpo do laço

# Exemplo

```
void main()
{ int cont, totPal; char letra;
  printf("Digite uma frase:\n");
  cont = 0;  totPal = 0;
  while( (letra=getche()) != '.' )
  { cont++;
    if(letra==' ') totPal++;
  }
  if(cont>0) totPal++;
  printf("\n");
  printf("Total de caracteres: %d\n",cont);
  printf("Total de palavras: %d\n",totPal);
}
```

## O laço *do/while*

- É semelhante ao while, porém testando a condição após a execução do laço. Dessa forma, o corpo do laço sempre é executado pelo menos uma vez.

```
do  
{ comandos  
}while(expressão de teste); _____
```

ponto-e-vírgula aqui

# Exemplo

```
int num;  
do  
{  
    printf("Digite um número para calcular ");  
    printf("seu fatorial. Digite um número ");  
    printf("negativo para finalizar: ");  
    scanf("%d",&num);  
    printf("%d! = %d\n",num,fat(num));  
  
} while(num>=0);
```

# do-while

- Estimativas indicam que o laço do-while é necessário em apenas 5% dos laços
  - ler a expressão de teste antes de percorrer o laço ajuda o leitor a interpretar o sentido do bloco de instruções



# O comando `break`

- A execução do comando `break` causa a saída imediata do laço
  - O comando `break` pode ser usado no corpo de qualquer laço
  - Se o `break` estiver em laços aninhados, afetará somente o laço mais interno
  - deve ser evitado pois pode causar dificuldade de leitura e confusão ao se manter o programa

# O comando `continue`

- O comando `continue` força a próxima iteração do laço e pula o código que estiver abaixo

# Exemplo

```
int num;
while (1)
{ printf("\n Digite um número maior que 0:");
  scanf("%d",&num);
  if(num < 0)
  { printf("número errado\n");
    continue;
  }
  printf("Número correto");
  if(num > 100)
    break;
}
```

# Exercício

- Faça um algoritmo que lê um conjunto de dados para candidatos a emprego (cpf, idade, sexo[m,f] e experiencia [s,n])
  - (cpf=0 indica fim dos candidatos)
- Ao fim da entrada de dados, escreva na tela o numero de candidatos homens, o numero de candidatos mulheres e idade média dos homens que já têm experiência no serviço
- Imprimir o percentual de mulheres acima de 30 anos sem experiência, dentre as mulheres sem experiência.
- Imprimir o percentual de pessoas com experiência.

# Exercício

- População país A: 9 mil, crescendo a 3% ao ano
- População país B: 20 mil, crescendo a 1,5% ao ano
- Fazer algoritmo que calcule e escreva o número de anos necessárias para a população do país A ultrapasse a população do país B

# Exercícios

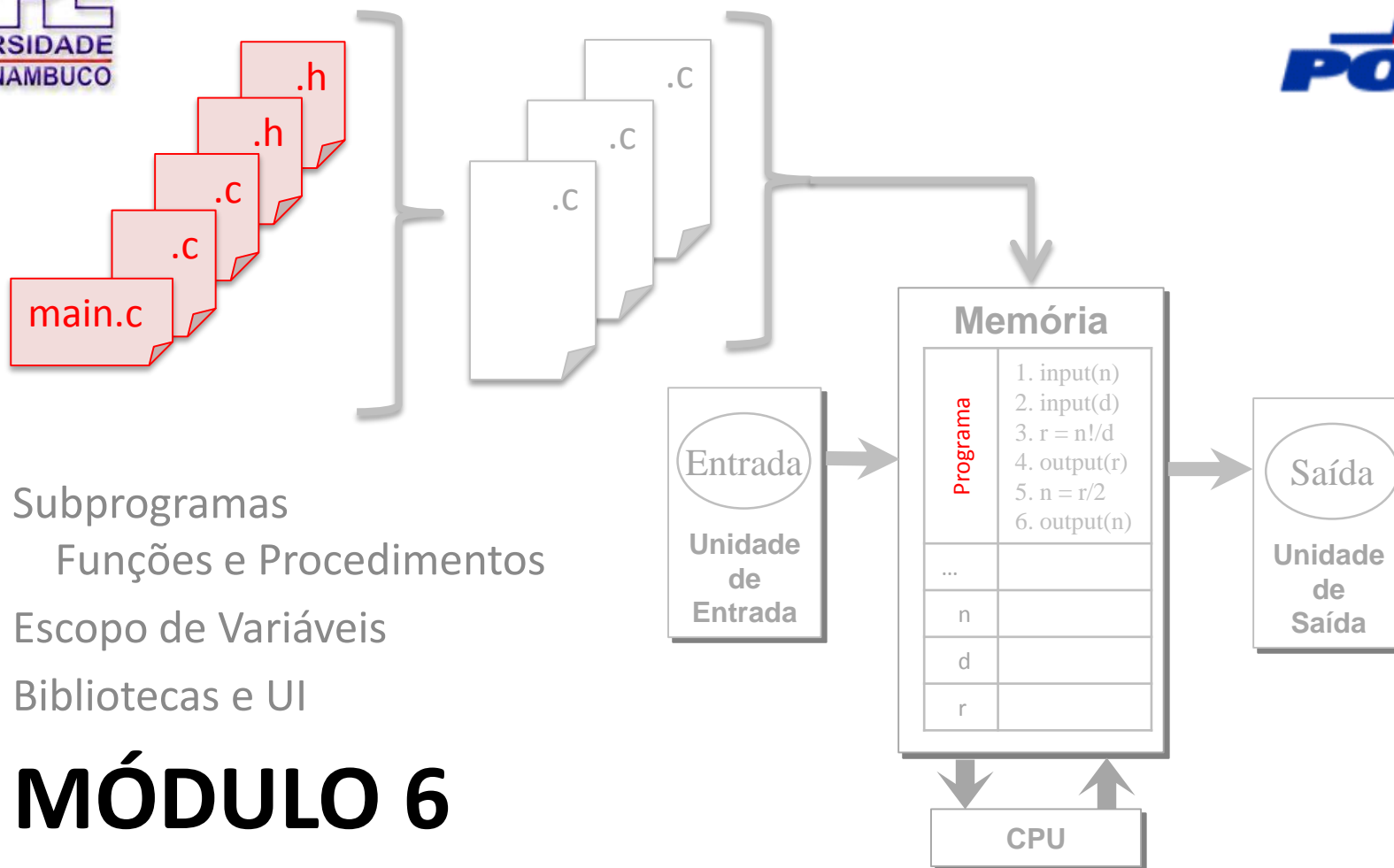
- Fazer o programa para calcular e imprimir os seguintes somatórios

$$2^1/50 + 2^2/49 + 2^3/48 + \dots + 2^{50}/1$$

$$1/1 - 2/4 + 3/9 - 4/16 + \dots - 20/400$$

# Exercícios

- A série de Fibonacci  
**0, 1, 1, 2, 3, 5, 8, 13, 21**
- Escreva um programa em C que imprima os **n** primeiros termos desta série (dados pelo usuário)





# Função

- Unidade autônoma do programa desenvolvida para executar alguma atividade
- Exemplos:
  - `getch()`
  - `printf()`
  - `scanf()`
  - `getchar()`
  - `rand()`

# Criando Funções

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

```
void main( )
```

```
{
```

```
    int cont;
```

```
    for(cont=1;cont<=19;cont++) printf("-");
```

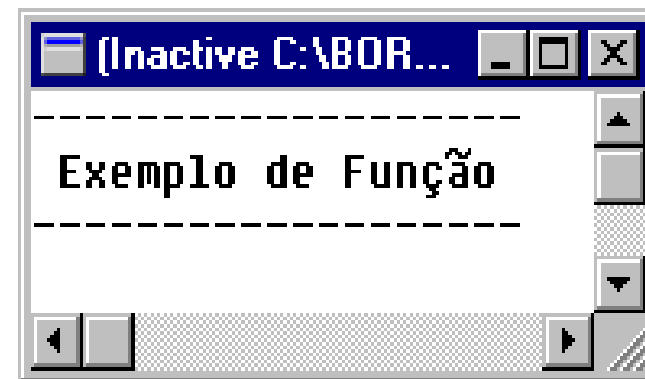
```
    printf("\n");
```

```
    printf(" Exemplo de Função \n");
```

```
    for(cont=1;cont<=19;cont++) printf("-");
```

```
    printf("\n");
```

```
}
```

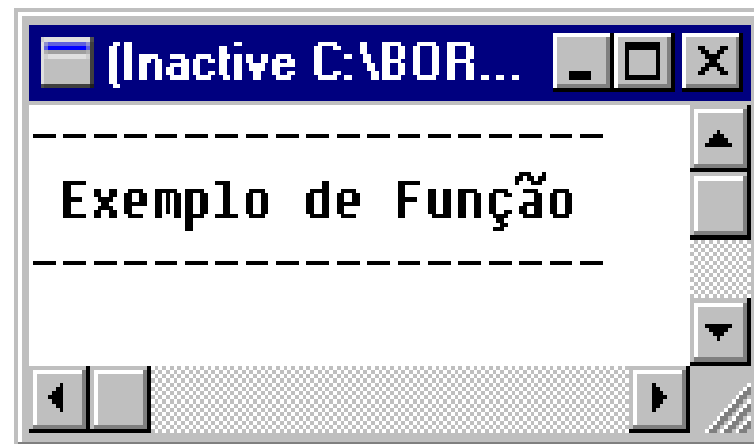


# Criando Funções

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

```
void linha( )  
{ int cont;  
  for(cont=1;cont<=19;cont++) printf("-");  
  printf("\n");  
}
```

```
void main( )  
{ linha( );  
  printf(" Exemplo de Função \n");  
  linha( );  
}
```



# Funções sem retorno (Procedimentos)

```
void tabelaASCII()  
{  
    unsigned char c;  
    printf("Código   Letra\n");  
    printf("-----\n");  
    for(c = 32; c < 255; c++)  
        printf(" %4d      %c\n", c, c);  
}
```

# Funções

## que retornam valores

```
float percentagem (float valor, float taxa)
{
    float resultado = valor * taxa/100;
    return resultado;
}
```

OU

```
float percentagem (float valor, float taxa)
{
    return valor * taxa/100;
}
```

# Funções recursivas

```
int fat(int n)
```

```
{
```

```
    if (n<=0)
```

```
        return 1;
```

```
    else
```

```
        return n*fat(n-1);
```

```
}
```

Caso base



Chamada recursiva!!



# Escopo de Variáveis

```
#include <stdio.h>

int gTotal;

void linha(char sep)
{
    int cont;
    for(cont=1;cont<=19;cont++) printf("%c",sep);
    gTotal = cont;
}

void main( )
{
    gTotal = 3;
    linha('=');
    printf("\nTotal = %d\n",gTotal);
}
```

Diagram illustrating variable scope:

- `#include <stdio.h>` is labeled **global**.
- `int gTotal;` is labeled **global**.
- `void linha(char sep)` is labeled **local**.
- `int cont;` is labeled **local**.

# Escopo de Variáveis

```
#include <stdio.h>
int total;
void linha(char sep)
{ int cont;
  for(cont=1;cont<=19;cont++) printf("%c",sep);
  total = cont;
}
void main( )
{ int cont = 5;
  total = 3;
  linha('=');
  printf("\nTotal = %d\n",total+cont);
}
```

variáveis diferentes!!



# Classe de armazenamento de variáveis

- **extern**
  - Cria variáveis globais em todos os programas, mas não duplica os rótulos

## Arquivo 1

```
int x,y;  
int funcao( )  
{ ...  
    x = 25;  
}
```

## Arquivo 2

```
extern int x,y;  
main( )  
{ ...  
    x = 2;  
}
```

# Classe de armazenamento de variáveis

- static
  - Variáveis locais que armazenam seu valor em várias execuções da mesma função
  - Evita uso de variáveis locais

```
int funcao( )  
{  
    static int a= 100;  
    a = a + 23;  
    return a;  
}
```

# Exercício

- Escreva uma função recursiva em C para calcular o *fibonacci* de um número
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Escreva um programa em C que use a função definida anteriormente para imprimir o *fibonacci* de um número indicado pelo usuário

# Exemplo – arquivos separados

- Criar um novo projeto no CodeBlocks
- Em um arquivo chamado `func.c`, crie duas funções:
  - `int quad(int x)` que retorna o quadrado de um número
  - `int cubo(int x)` que retorna o cubo de um número

# Resposta

```
int quad(int x)
{ return x*x; }
```

```
int cubo(int x)
{ return x*x*x; }
```

# Assinaturas – arquivo h

- No mesmo projeto, em um arquivo chamado `func.h`, crie assinaturas das três funções

```
int quad(int x);
```

```
int cubo(int x);
```

# Exemplo

- Agora crie um arquivo chamado `ex01.c` que contenha a função `main()` que irá:
  - Solicitar um número `n` ao usuário e imprimir o quadrado de todos os números de 1 a `n` e o cubo de todos os números de 1 a `n`
  - Use as funções que estão no arquivo `func.c`

# Resposta

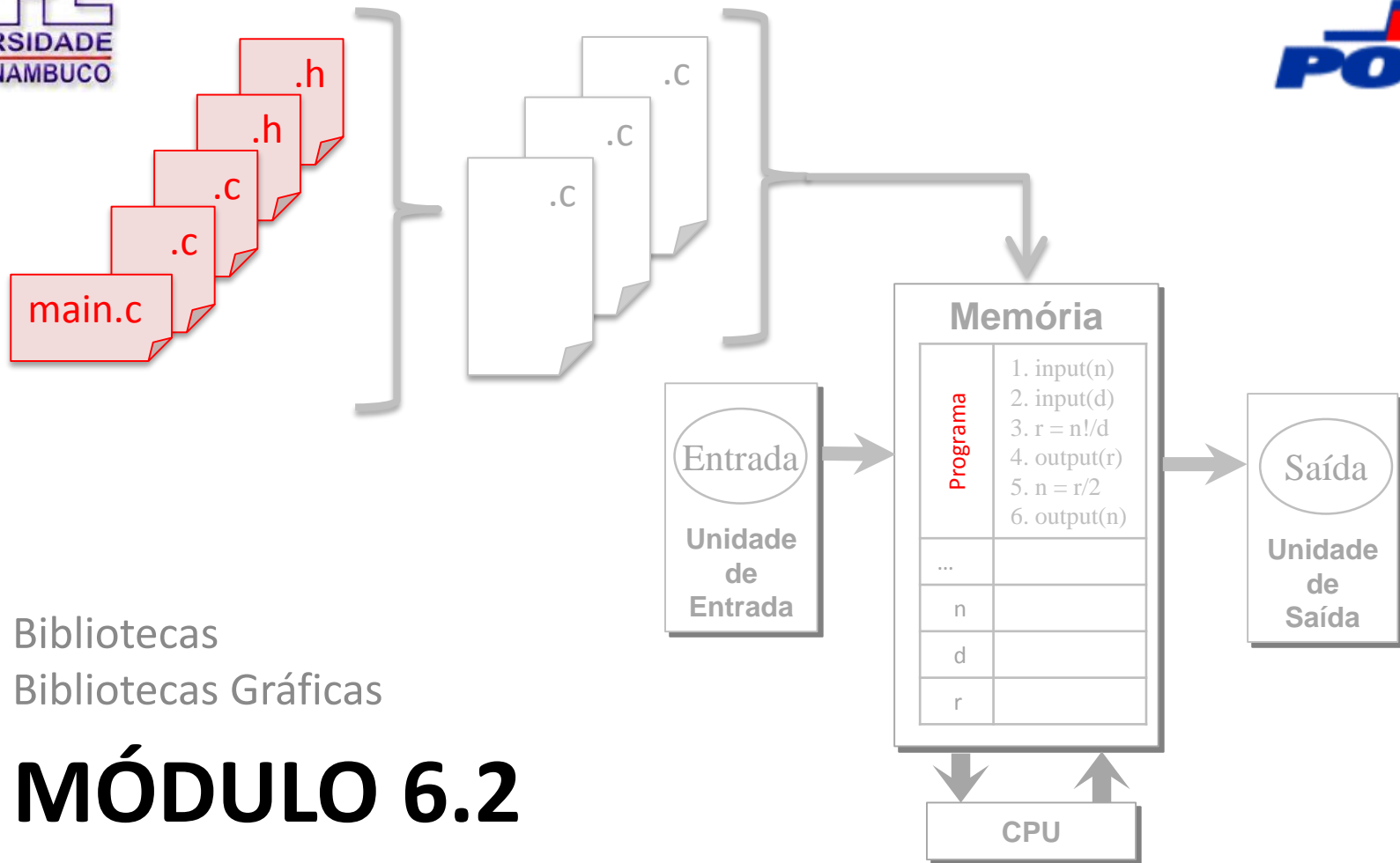
```
#include <stdio.h>
#include "func.h"
```

```
void main()
{ ...
}
```



Dica: criar um projeto  
incluindo os 3 arquivos





Bibliotecas  
Bibliotecas Gráficas

## MÓDULO 6.2

# Evolução da Linguagem C

- Links:
  - <http://www.open-std.org/>
  - <http://www.open-std.org/JTC1/SC22/>
  - Linguagem C
    - <http://www.open-std.org/JTC1/SC22/WG14/>
      - <http://www.open-std.org/JTC1/SC22/WG14/www/standards.html#9899>
      - <http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf>
    - [http://www.iso.org/iso/iso\\_catalogue/catalogue\\_tc/catalogue\\_detail.htm?csnumber=57853](http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853)

# Links Úteis

- C Frequently Asked Questions
  - <http://c-faq.com/>
  - <http://c-faq.com/questions.html>
- Apostila C UFMG
  - [http://sistemas.riopomba.ifsudestemg.edu.br/dcc/materiais/23854759\\_Apostila%20Linguagem%20C.pdf](http://sistemas.riopomba.ifsudestemg.edu.br/dcc/materiais/23854759_Apostila%20Linguagem%20C.pdf)

# Evolução da Linguagem C++

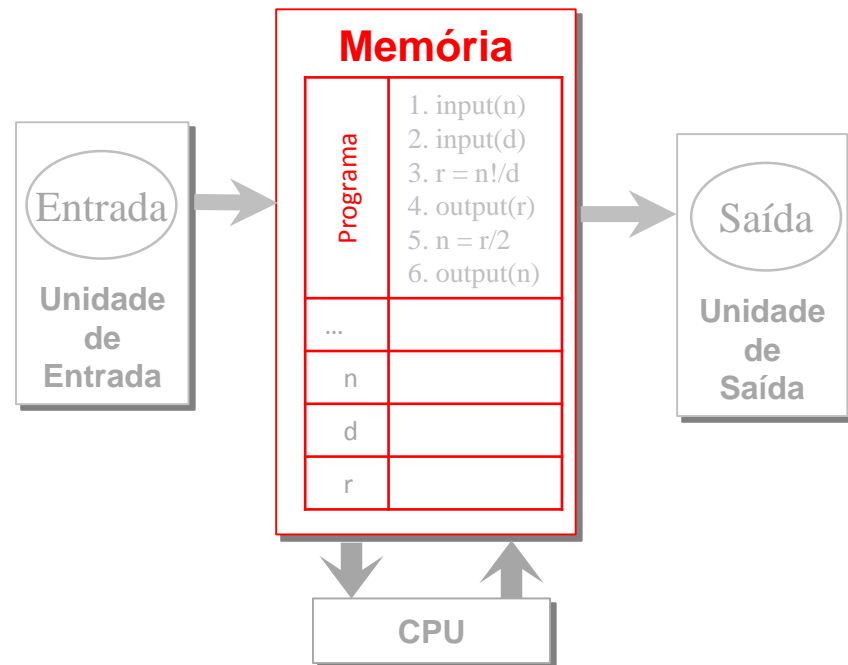
- Links:
  - <http://www.open-std.org/>
  - <http://www.open-std.org/JTC1/SC22/>
  - Linguagem C++
    - <http://www.open-std.org/JTC1/SC22/WG21/>
    - <http://isocpp.org/std>
      - <https://isocpp.org/files/papers/N3690.pdf>
    - Bibliotecas C dentro de C++
      - <http://www.cplusplus.com/reference/clibrary/>

# *Graphical User Interface (GUI)*

- Interface **Gráfica** com o Usuário
- Exemplos de Bibliotecas Gráficas:
  - Allegro (<http://alleg.sourceforge.net/>)
  - Qt (<https://qt-project.org/>)
  - Gtk (<http://www.gtk.org/>)
  - Win API ([http://msdn.microsoft.com/en-us/library/ff818516\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ff818516(v=vs.85).aspx))
    - Win32

Vetores e Matrizes  
Strings

# MÓDULO 7



# Exemplo: média de notas

- Observe:

```
int nota1, nota2, nota3;
printf("Digite a nota do aluno 1: ");
scanf("%d",&nota1);
printf("Digite a nota do aluno 2: ");
scanf("%d",&nota2);
printf("Digite a nota do aluno 3: ");
scanf("%d",&nota3);
printf("A média dos alunos é: %.2f",
      (nota1+nota2+nota3)/3.0);
```

# Vetores

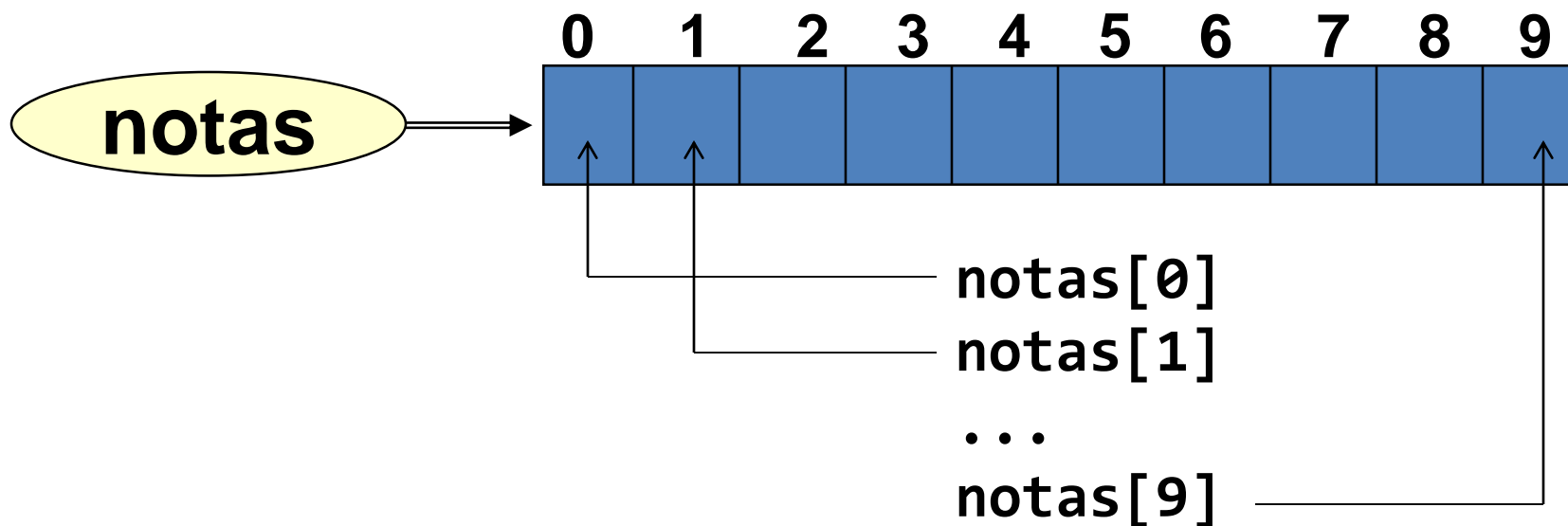
- O que acontece se você desejar encontrar a média dos 2000 alunos de uma escola?
- Solução: usar um vetor com 2000 itens
  - Um vetor é um tipo de dado utilizado para representar um conjunto de valores homogêneos utilizando um único nome.
  - Cada valor é diferenciado através do índice do vetor
  - Em C, o primeiro elemento tem índice 0.



# Vetores

```
int notas[10];
```

- Aloca 10 valores inteiros referenciados através do identificador notas.



# Exemplo

```
int notas[2000];  
int soma, i;  
for(i = 0; i < 2000; i++)  
{ printf("Digite a nota do aluno %d: ",i);  
  scanf("%d",&notas[i]);  
}  
soma = 0;  
for(i = 0; i < 2000; i++)  
  soma = soma + notas[i];  
printf("Média = %.2f\n",soma/2000.0);
```

# Cuidados

- C **não** avisa quando o limite de um vetor é excedido!
  - Se o programador transpuser o fim do vetor durante a operação de atribuição, os valores serão armazenados em outros dados ou mesmo no código do próprio programa.
- O programador tem a responsabilidade de verificar o limite do vetor

# Inicializando Vetores

```
int tab[5] = {7, 0, -9, 15, 38};
```

```
int fib[] = {1,1,2,3,5,8,13,21,34};
```

- Na ausência do tamanho do vetor, o compilador contará o total de itens na lista de inicialização
- Se o programador declarar um vetor sem inicializá-lo, ele deverá declarar sua dimensão
- Se fizer os dois
  - Se há menos inicializadores que a dimensão especificada, os outros serão zero
  - Mais inicializadores que o necessário implica em warning

# Vetores e matrizes

- Aprendemos a declarar vetores
- `int notas[10];`
- Mas podemos usar vetores de mais de uma dimensão – MATRIZ

```
int matriz1[3][3]; ([linhas][cols])  
int matriz2[][2]={ {1,2}, {5,6} };  
int matriz3[2][2]={0,1,2,3};
```

# Percorrendo matrizes

```
int i,j;  
int matriz[2][2]={{1,2},{5,6}};  
for (i=0;i<2;i++)  
{  
    for (j=0;j<2;j++)  
    {  
        printf("%d ",matriz[i][j]);  
    }  
    printf("\n");  
}
```

# Exercício

- Crie um programa que leia um vetor de 10 números inteiros, que
- calcula e imprime, depois de ler todos os números
  - A soma dos números
  - A multiplicação dos números nas posições pares
  - A média ponderada dos números (peso = posição do número no vetor)
  - O vetor normalizado
    - $v[i] \leftarrow (v[i] - \min) / (\max - \min)$

# Strings

- Podemos armazenar uma sequência de caracteres (string) em um vetor:

```
char nome[5];
```

- Como C não controla automaticamente o limite do vetor, sempre devemos sinalizar o final do string com o caractere especial '`\0`'

```
nome[0] = 'c';   nome[1] = 'a';
```

```
nome[2] = 's';   nome[3] = 'a';
```

```
nome[4] = '\0';
```



# Strings constantes

```
printf("Um string constante!\n");  
printf("%s fica muito longe", "Plutão");
```

## ERRADO

```
char nome[10] = {"Corrida"};  
nome = "Viagem";
```

## CORRETO

```
char nome[10] = "Corrida";  
char nome[10] =  
    {'C', 'o', 'r', 'r', 'i', 'd', 'a', '\0'};
```

# Funções para Strings

- Definidas em `string.h`:
  - `strcpy(char *destino, char *fonte);`
  - `strcat(char *destino, char *fonte);`
  - `strcmp(char *str1, char *str2);`
    - retorna um inteiro positivo se *s1* é lexicamente posterior que *s2*; zero se as duas são idênticas; e negativo se *s1* é lexicamente anterior que *s2*

# Funções para Strings

- Definidas em `string.h`:
  - `strlen(char *fonte);`
  - `sprintf(char *destino, char *controle, ...);`
  - `gets(char *destino);`
  - `puts(char *fonte);`

# Exemplo

```
char nome[21];  
int ano[2];  
printf("Qual é seu nome? ");  
gets(nome);  
printf("%s, em que ano estamos? ",nome);  
scanf("%d", &ano[0]);  
printf("%s, em ano você nasceu? ",nome);  
scanf("%d",&ano[1]);  
printf("%s, sua idade é %d anos.\n",nome,ano[0]-ano[1] );
```

## Exemplo – fazer para ver o resultado

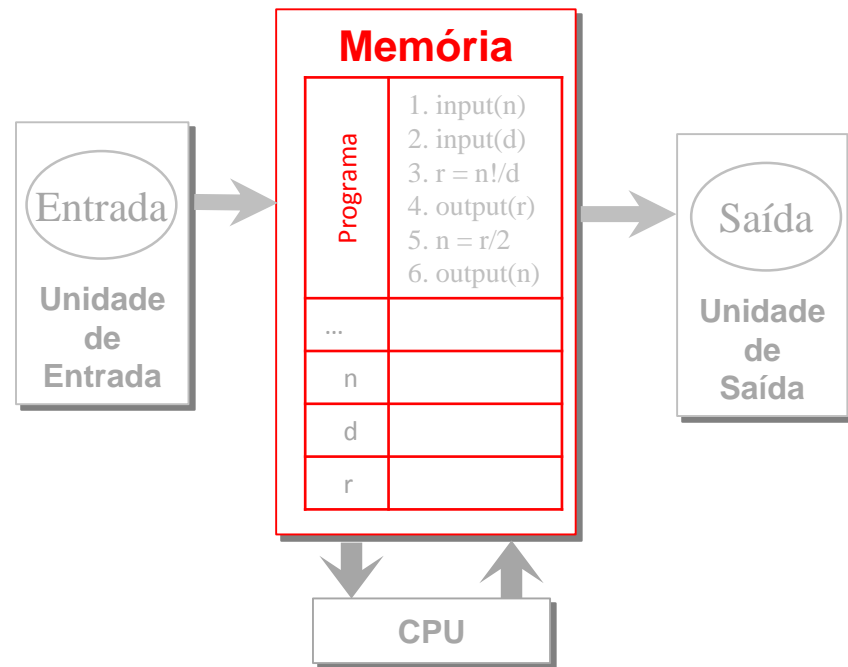
```
char msg[81], nome[21], sobrenome[21];  
int idade;  
printf("Qual é seu nome? ");  
gets(nome);  
printf("Qual é seu sobrenome? ");  
gets(sobrenome);  
printf("Qual é sua idade? ");  
scanf("%d", &idade);  
strcpy(msg,nome);  
strcat(msg," ");  
strcat(msg,sobrenome);  
sprintf(msg,"%s tem %d anos de idade",msg,idade);  
puts(msg);
```

# Exercício

- Fazer um programa que pede uma string ao usuário
- Imprimir na tela os caracteres da string entrada juntamente com seus códigos ASCII, um abaixo do outro

Estruturas de dados  
Definição de Tipos

# MÓDULO 8



# Estruturas

- Agrupa conjunto de tipos de dados distintos sob um único nome
- Chamadas de Registros

## Cadastro Pessoal

string	Nome
string	Endereço
inteiro	Telefone
inteiro	Idade
inteiro	Data de Nascimento
float	Peso
float	Altura

```
struct cadastro_pessoal {  
    char    nome[50];  
    char    endereço[100];  
    int      telefone;  
    int      idade;  
    int      nascimento;  
    float    peso;  
    float    altura;  
};
```



# Estruturas

- Um pequeno exemplo

```
main( )  
{  
    struct facil {  
        int num;  
        char ch;  
    };  
  
    struct facil x; /*declara variável x do tipo facil */  
    x.num = 2;  
    x.ch = 'Z';  
    printf("x.num = %d, x.ch = %c\n", x.num, x.ch);  
}
```

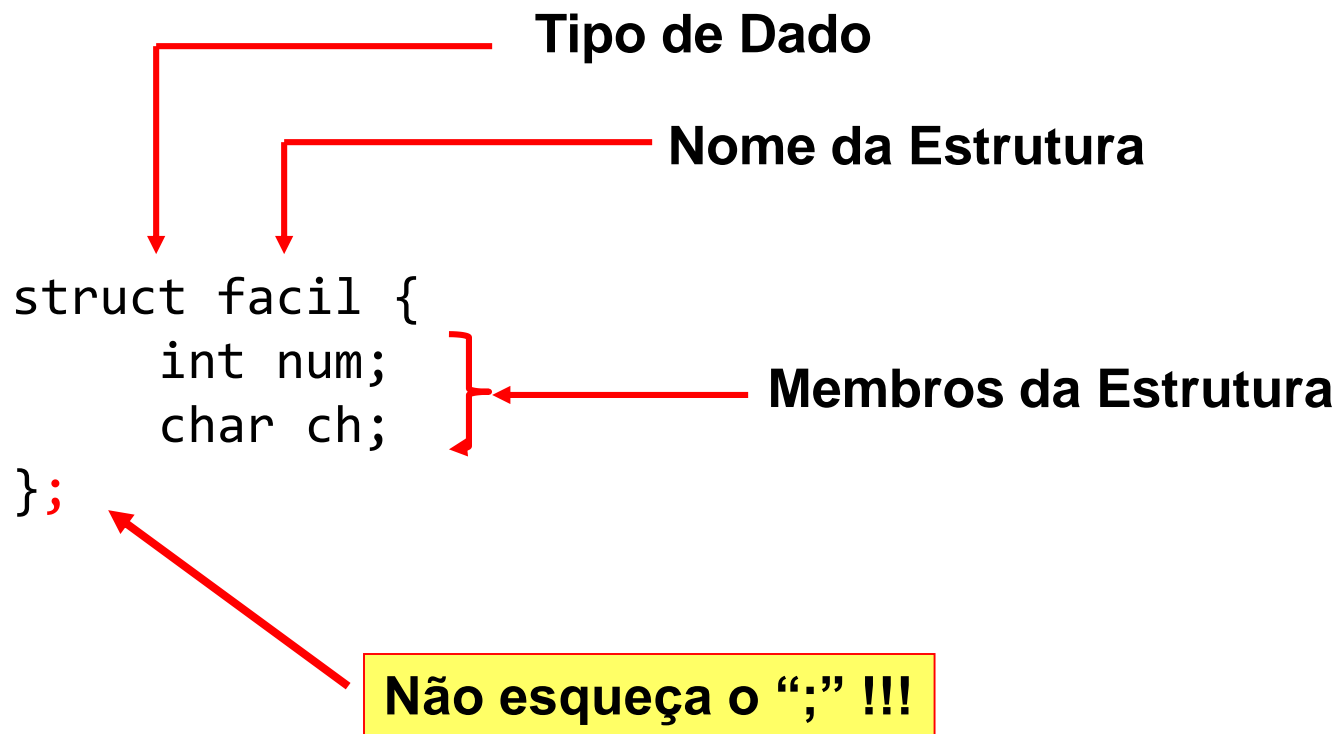
# Estruturas

- Outra forma...

```
main( )  
{  
    struct facil {  
        int num;  
        char ch;  
    } x;    /*declara variável x do tipo facil */  
  
    x.num = 2;  
    x.ch   = 'Z';  
    printf("x.num = %d, x.ch = %c\n", x.num, x.ch);  
}
```

# Estruturas

- Análise



# Exemplo

```
main( )
```

```
{
```

```
    struct facil {
```

```
        int num;
```

```
        char ch;
```

```
    };
```

Mais de  
uma  
variável

```
    struct facil x1; /*declara variável x1 do tipo facil */
```

```
    struct facil x2; /*declara variável x2 do tipo facil */
```

```
    x1.num = 2;    x1.ch    = 'Z';
```

```
    x2.num = 3;    x2.ch    = 'B';
```

```
    printf("x1.num = %d, x1.ch = %c\n", x1.num, x1.ch);
```

```
    printf("x2.num = %d, x2.ch = %c\n", x2.num, x2.ch);
```

```
}
```

# Exemplo

- Outra opção para definição das variáveis:

```
main( )  
{  
    struct facil {  
        int num;  
        char ch;  
    } x1, x2;  
  
    x1.num = 2;    x1.ch    = 'Z';  
    x2.num = 3;    x2.ch    = 'B';  
    printf("x1.num = %d, x1.ch = %c\n", x1.num, x1.ch);  
    printf("x2.num = %d, x2.ch = %c\n", x2.num, x2.ch);  
}
```

# Estruturas

- Pode-se definir uma estrutura sem um nome...

```
struct {  
    int num;  
    char ch;  
} x1, x2;
```

# typedef

- Podemos criar novos tipos com apenas um nome, usando o comando typedef

```
main( )  
{  
    typedef struct facil {  
        int num;  
        char ch;  
    } tFacil;  
    // Ou usaria: typedef struct facil tFacil;  
    tFacil x1, x2;  
    x1.num = 2;    x1.ch = 'Z';  
    x2.num = 3;    x2.ch = 'B';  
    printf("x1.num = %d, x1.ch = %c\n", x1.num, x1.ch);  
    printf("x2.num = %d, x2.ch = %c\n", x2.num, x2.ch);  
}
```

Útil para structs

# Estruturas

- Criando uma Lista de Livros
  - Passo inicial: 2 Livros

```
struct livro {  
    char titulo[40];  
    int  regnum;  
};
```

```
struct livro livro1 =  
    {"Treinamento em Linguagem C - I", 124};  
struct livro livro2 =  
    {"Treinamento em Linguagem C - II", 125};
```



# Atribuições e Estruturas

- Na versão original do C, definida por Kernighan e Ritchie, era impossível atribuir o valor de uma variável estrutura a outra do mesmo tipo usando uma simples expressão de atribuição.
- Nas versões modernas de C, esta forma de atribuição já é possível

# Exemplo

```
struct livro {  
    char titulo[40];  
    int  regnum;  
};
```

```
struct livro livro1 =  
    {"Treinamento em Linguagem C - I", 124};  
struct livro livro2 =  
    {"Treinamento em Linguagem C - II", 125};
```

```
livro2 = livro1;
```

# Estruturas Aninhadas

- Exatamente como é possível ter vetores de vetores, pode-se criar estruturas que contêm outras estruturas.

```
struct professor {  
    char nome[50];  
    char disciplina[20];  
    int  carga_horaria;  
};  
struct aluno {  
    char nome[50];  
    int matricula;  
};  
struct cadastro_escolar {  
    struct professor docentes[100];  
    struct aluno* discentes;  
};
```

# Exercício

- Considere que foi definida a seguinte estrutura:

```
struct    fracao {  
    int numerador, denominador;  
};
```

- Escreva um programa em C que calcule as quatro operações usando frações definidas com esta estrutura. O programa deve ler duas frações e imprimir o resultado de cada uma das quatro operações

Unions

Enums

Bit Sets

# ESTRUTURAS AVANÇADAS

# Unions

- Uma declaração union determina uma **única** localização de memória onde podem estar armazenadas várias variáveis diferentes.

```
union angulo {  
    float graus;  
    float radianos;  
};  
  
void main() {  
    union angulo ang;  
    char op;  
    printf("\nGraus ou radianos (G/R)?");  
    scanf("%c",&op);  
    printf("\nAngulo:");  
    if (op == 'G'){  
        ang.graus = 180;  
        printf("%d\n",ang.graus);  
    } else if (op == 'R') {  
        ang.radianos = 3.1415;  
        printf("%f\n",ang.radianos);  
    } else printf("\nInvalido!\n");  
}
```

# Enumerações (Enums)

- Numa enumeração podemos dizer ao compilador quais os valores que uma determinada variável pode assumir. Sua forma geral é:

```
enum nome_do_tipo_da_enumeração {  
    lista_de_valores  
} lista_de_variáveis;
```

- Cada valor está associado a um número, começando em 0. Logo as variáveis são do **tipo int**.

# Enumerações (Enums)

- Exemplo:

```
enum dias_da_semana {segunda, terca, quarta, quinta, sexta, sabado,
domingo};
void main ()
{
    enum dias_da_semana d1 = segunda, d2 = sexta;
    if (d1 == d2)
        printf ("O dia e o mesmo.");
    else
        printf ("São dias diferentes.");
    return 0;
}
```



# Bit Fields

- Útil quando temos pouca memória
- Permite compactar os dados da estrutura, isto é, compactar vários valores em apenas uma palavra (word) da máquina

```
struct pacote {  
    unsigned int f1:1;  
    unsigned int f2:1;  
    unsigned int f3:1;  
    unsigned int f4:1;  
    unsigned int type:4;  
    unsigned int my_int:9;  
} p;
```

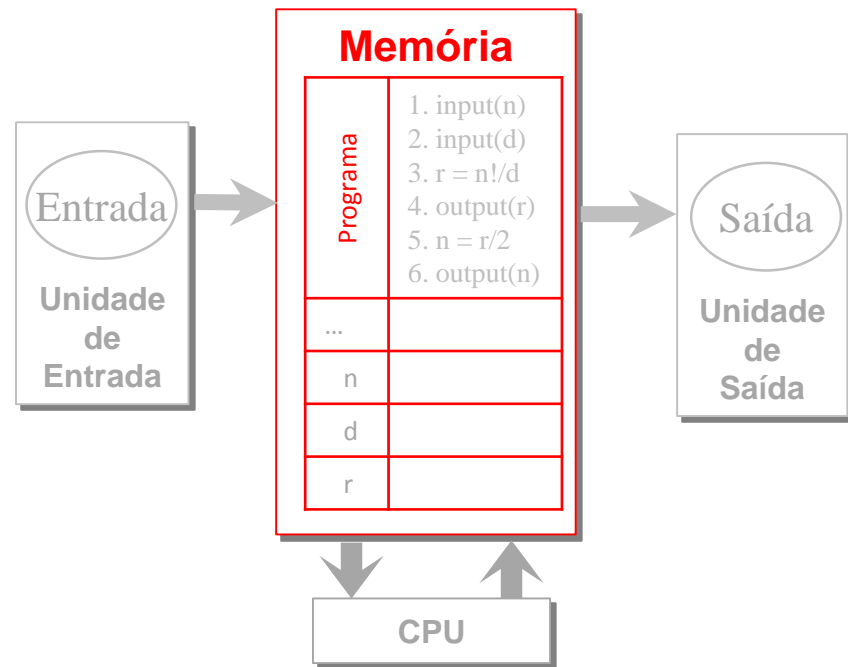
**Observe** que após cada : há um número de bits desejado para o membro da estrutura. Assim, neste exemplo, a variável p ocupará aproximadamente 17 bits.

- Em cada computador poderá ter um pouco mais de bits dependendo do tamanho de uma word.

Ponteiros

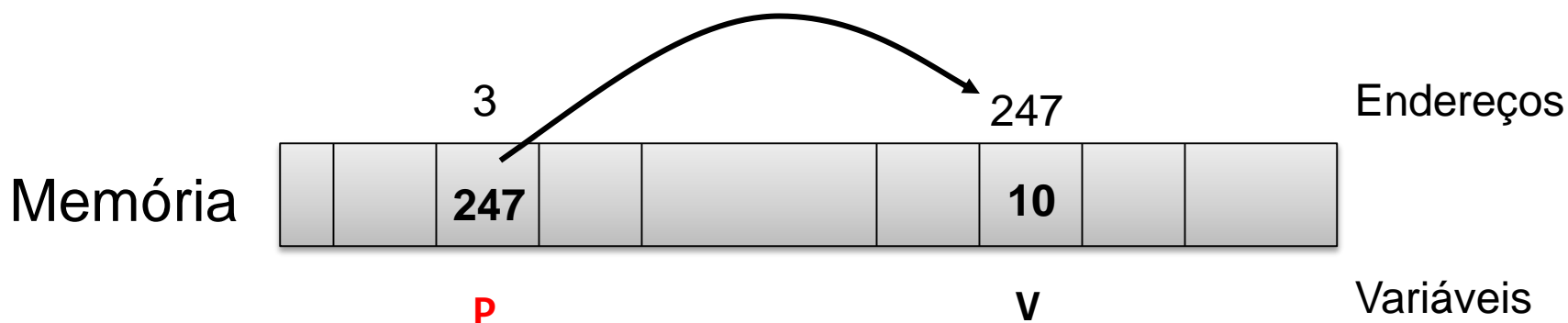
Alocação de Memória

# MÓDULO 9



# Ponteiros

- Um ponteiro é uma variável que contém o endereço de outra variável



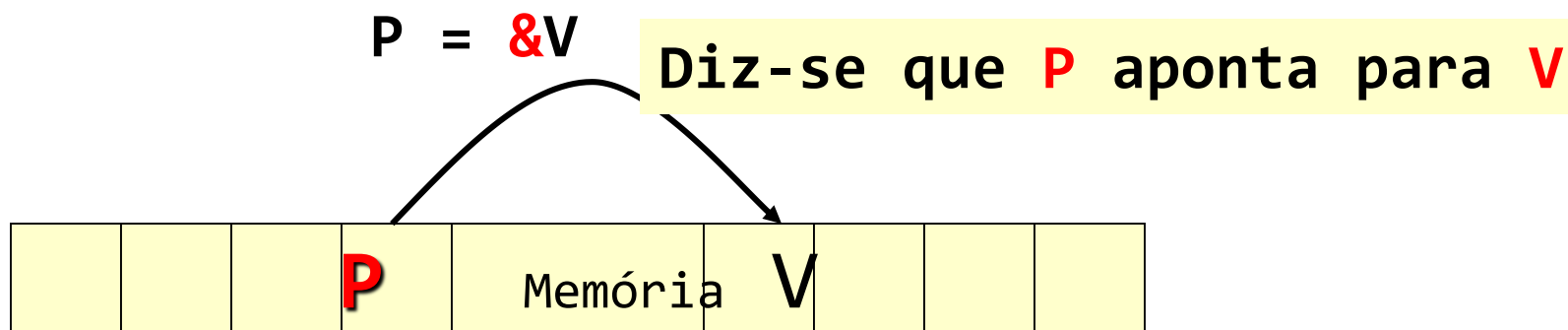
**P** = endereço da variável V

# Ponteiros – MOTIVAÇÃO

- Possibilitar que funções modifiquem os argumentos que recebem
- Manipular vetores e strings - útil para passar vetores como parâmetro
- Criar estruturas de dados mais complexas, como listas encadeadas, árvores binárias etc.
- Código mais eficiente (compilam mais rápido)

# Operador &

- Fornece o endereço de um objeto



- Só se aplica a objetos na memória: variáveis e vetores
- Não pode ser aplicado a expressões ou constantes
  - Ex:  $x = \&3;$

**ERRADO**

# Operador de Indireção \*

- Quando aplicado a um ponteiro, acessa o objeto apontado por ele

```
int x=1, y=2, z[3];
int *ip; /* ip é um ponteiro para int */
ip=&x; /*ip aponta para o endereço de x*/
y=*ip; /*y = valor de ip (ou seja, 1)*/
*ip=0;
ip=&z[0];
```

x	0
y	1
ip	
z[0]	?
z[1]	?
z[2]	?
⋮	⋮

# Ponteiros

- Suponha o código abaixo:

$x=5;$

$y=x;$

$x=x+1;$

- Como atualizar o valor de Y sempre que X for alterado para que *sempre*  $X = Y$  ???

- Solução 1: Qualquer alteração em X deve ser seguida pela linha  $Y=X$
- Solução 2: Ponteiros !!

# Ponteiros

```
int x, *y;  
x = 5;  
y = &x;  
x++;  
printf ("X = %d, Y= %d\n", x, *y);
```

**Y mantém o mesmo  
valor de X !!**



# Ponteiros

```
int *p1,*p2, i, j;
```

```
i=1;
```

```
p1=&i;
```

```
p2=p1;
```

```
i=3;
```

```
j=4;
```

```
p1=&j;
```

```
printf("*p1=%d, *p2=%d, i=%d, j=%d\n\n",*p1,*p2,i,j);
```

```
printf("p1=%d, p2=%d, &i=%d, &j=%d\n",p1,p2,&i,&j);
```

Qual o resultado? (fazer no papel, sem usar o computador)  
Usar desenhos.  
O valor dos endereços podem ser fictícios.  
Ex: posição x, y, z, w da memória.

# Operações com Ponteiros

```
main( )  
{
```

```
    int x=5, y=6;  
    int *px, *py;
```

```
    px = &x;  
    py = &y;
```

Resultado: 1

4 bytes de diferença

```
    if (px<py) printf("py-px = %u\n",py-px);  
    else      printf("px-py = %u\n",px-py);
```

```
    printf("px = %u, *px = %d, &px = %u\n", px, *px, &px);  
    printf("py = %u, *py = %d, &py = %u\n", py, *py, &py);  
    py++;  
    printf("py = %u, *py = %d, &py = %u\n", py, *py, &py);  
    py=px+3;  
    printf("py = %u, *py = %d, &py = %u\n", py, *py, &py);  
    printf("py-px = %u\n",py-px);
```

```
}
```

# Operações com Ponteiros

$px - py = 1$

$px = 65488, *px = 5; \&px = 65460$

$py = 65484, *py = 6; \&py = 65464$

$py = 65488, *py = 5; \&py = 65464$

$py = 65500, *py = ?; \&py = 65464$

$py - px = 3$

Testes relacionais

$\geq, \leq, <, >, ==$

são **aceitos** em ponteiros

A diferença entre dois ponteiros será dada na unidade do **tipo de dado** apontado

# Operações com Ponteiros

- O incremento de um ponteiro acarreta na movimentação do mesmo para o próximo valor do **tipo apontado**
  - Ex: Se **px** é um ponteiro para int com valor 3000, depois de executada a instrução **px++**, o valor de **px** será 3004 e não 3001 !!!
    - Obviamente, o deslocamento varia de compilador para compilador dependendo do número de bytes adotado para o referido tipo

# Endereços como

## Argumentos para uma Função

- Como uma função pode alterar variáveis de quem a chamou?
  1. função chamadora passa os endereços dos valores que devem ser modificados
  2. função chamada deve declarar os endereços recebidos como ponteiros

# Endereços como Argumentos para uma Função

## Chamada por valor

```
main( )
{
    int x, y;
    x=0;
    y=0;
    altera2(x,y);
    printf("1º é %d, 2º é  
%d.",x, y);
}

void altera2(int px, int py)
{
    px = 3;
    py = 5;
}
```

**NÃO** altera os  
valores de x e y

## Chamada por referência

```
main( )
{
    int x, y;
    x=0;
    y=0;
    altera2(&x,&y);
    printf("1º é %d, 2º é %d.",x,y);
}

void altera2(int *px, int *py)
{
    *px = 3;
    *py = 5;
}
```

Altera os valores  
de x e y

# Endereços como Argumentos para uma Função

```
main( )  
{  
    int x, y;  
    x=0;  
    y=0;  
    altera2(&x,&y);  
    printf("O 1o. é %d, o 2o. é %d.", x, y);  
}  
  
void altera2(int *px, int *py)  
{  
    *px = 3;  
    *py = 5;  
}
```

**\*px e \*py são do tipo int**

**px e py contém endereços  
de variáveis do tipo int**

# Exercício

- Escreva um programa que aplica a função **exponencial\_2** a uma variável inteira e imprime o resultado da aplicação.
- A função **exponencial\_2** deve ser do tipo void e eleva um número ao quadrado
- O resultado deve ser armazenado na própria variável inteira passada como parâmetro para **exponencial\_2**



# Ponteiros, Vetores e Matrizes

- Em C existe um relacionamento muito forte entre ponteiros e vetores
  - O compilador transforma todo vetor e matriz em ponteiros, pois a maioria dos computadores é capaz de manipular ponteiros e não vetores
  - Qualquer operação que possa ser feita com índices de um vetor pode ser feita com ponteiros
  - O nome de um vetor é um endereço, ou seja, um ponteiro

# Ponteiros e Vetores

## Versão com Vetor

```
main( )
{
    int nums[ ] = {1, 4, 8};
    int cont;

    for(cont=0; cont < 3; cont++)
        printf("%d\n", nums[cont]);
}
```

## Versão com Ponteiro

```
main( )
{
    int nums[ ] = {1, 4, 8};
    int cont;

    for(cont=0; cont < 3; cont++)
        printf("%d\n", *(nums + cont));
}
```

1  
4  
8

Endereço inicial do vetor

Deslocamento

# Ponteiros e Vetores

Observe a diferença para.....

```
main( )  
{  
    int nums[ ] = {1, 4, 8};  
    int cont;  
  
    for(cont=0; cont < 3; cont++)  
        printf("%d\n", (nums + cont));  
}
```

2293600
2293604
2293608

Sem o \* !!

# Ponteiros e Vetores

- Observação sobre o tamanho do vetor:
  - Comando *sizeof*

```
main()
{
    int num[ ]={1,2,3};

    printf ("Tamanho = %d\n", sizeof(num));
    printf ("Numero de elementos = %d\n", sizeof(num)/sizeof(int));
}
```

Observe o resultado !!

Observe o resultado !!

O que representam ?

# Ponteiros e Vetores

- Escreva um programa que lê um conjunto de, no máximo, 40 notas, armazena-as em um vetor e, por fim, imprime a média das notas.
  - Obs: utilize ponteiros para manipular o vetor
  - O programa pára de pedir as notas e prossegue com o cálculo da média quando o usuário entra com uma nota  $< 0$

# Ponteiros e Vetores

```
main( )  
{  
    float notas[40], soma=0;  
    int cont=0;  
    do {  
        printf("Digite a nota do aluno %d: ", cont);  
        scanf("%f", notas+cont);  
        if(*(notas+cont) > -1)  
            soma += *(notas+cont);  
    } while(*(notas+cont++) >= 0);  
  
    printf("Média das notas: %.2f", soma/(cont-1));  
}
```

# Ponteiros e Vetores

- Será que existe alguma maneira de simplificar a expressão `while (* (notas+cont++) > 0 )` ?

**`while (*(notas++) > 0)`**

**Errado !! “notas” é um ponteiro constante ! É o endereço do vetor notas e não pode ser trocado durante a execução do programa !**

**Apenas um ponteiro variável pode ser alterado**

# Ponteiros e Vetores

- Vamos re-escrever o programa anterior com um **ponteiro variável**....

```
main( )
{
    float notas[40], soma=0.0;
    int cont=0; float *ptr;
    ptr = notas;
    do {
        printf("Digite a nota do aluno %d: ", cont++);
        scanf("%f",ptr);
        if(*ptr > -1)
            soma += *ptr;
    } while(*(ptr++) > 0 && cont <= 40);
    printf("Média das notas: %.2f", soma/(cont-1));
}
```



# Vetor como Parâmetro (COM ponteiro)

Somando uma constante  
aos elementos de um vetor

```
#define TAM 5
adcons(int *ptr, int num, int cons)
{
    int k;
    for(k=0; k<num; k++) {
        *ptr = *ptr + cons;
        ptr++;
    }
}
main( )
{
    int vetor[TAM]={3,5,7,9,11};
    int c=10;
    int j;
    adcons(vetor, TAM, c);
    for(j=0; j<TAM; j++)
        printf("%d ", *(vetor+j));
}
```

# Vetor como Parâmetro (SEM ponteiro)

Uso do [ ], sem ponteiros

```
#define TAM 5
adcons(int ptr[], int num, int cons)
{
    int k;
    for(k=0; k<num; k++) {
        ptr[k] = ptr[k] + cons;
    }
}

main( )
{
    int vetor[TAM]={3,5,7,9,11};
    int c=10;
    int j;
    adcons(vetor, TAM, c);
    for(j=0; j<TAM; j++)
        printf("%d ",(vetor[j]));
}
```

# Ponteiro e Strings

- Um string é um vetor de caracteres
- Exemplo:

```
main( )  
{  
    char c='t';  
    char nome[10]="teste";  
  
    printf ("Texto1 = %c\n", c);  
    printf ("Texto2 = %s\n", nome);  
}
```

Observe a  
diferença  
na notação !

# Ponteiro e Strings

- Exemplo 2:
  - Modifique o programa anterior para...

```
main( )  
{  
    char c='t';  
    char nome[10]="teste de texto";  
  
    printf ("Texto1 = %c\n", c);  
    printf ("Texto2 = %s\n", nome);  
}
```

O que  
acontece ?

# Ponteiro e Strings

- Exemplo 3:
  - Modifique novamente para...

```
main( )  
{  
    char c='t';  
    char nome[10];  
    scanf("%s", nome);  
    printf("%c\n", c);  
    printf ("%s\n", nome);  
}
```

Digite:  
cddvfvfddfbggbgghnh  
(qualquer coisa  
com  
mais de 10  
caracteres)

# Ponteiro e Strings

```
main( )  
{  
    char *salute="saudacoes";  
    char nome[8];  
  
    puts("Digite seu nome");  
    gets(nome);  
    puts(salute);  
}
```

**Ponteiro variável**

`puts(++salute);`

audações (sem S)

```
main( )  
{  
    char salute[] = "saudacoes";  
    char nome[8];  
  
    puts("Digite seu nome");  
    gets(nome);  
    puts(salute);  
}
```

**Ponteiro constante**

`puts(++salute);`

**ERRO**

# Matriz de Caracteres (Vetor de strings)

```
char list[5][6];
```

...

	0	1	2	3	4	5	6
list[0] →	C	a	r	l	o	s	\0
list[1] →	A	n	a	\0			
list[2] →	P	e	d	r	o	\0	
list[3] →	A	n	d	r	e	\0	
list[4] →	S	i	l	v	i	a	\0

Desperdício  
de Memória

# Inicializando a Matriz

```
main( )  
{  
    int cont;  
    int entra=0;  
    char nome[40];  
    char list[5][7]=  
        { "Carlos", "Ana", "Pedro", "Andre", "Silvia" };  
    ...  
}
```

Versão multidimensional  
Cuidado com as  
dimensões da matriz !

Modifique o número  
de colunas para 6 e  
teste se o nome  
Roberto está incluído e  
se pode ser incluído.



# Exercício

## (Verificando se inclui um nome)

```
main( )
{
    int cont;
    int entra=0;
    char nome[40];
    char list[5][6]=
        { "Carlos", "Ana", "Pedro", "Andre", "Silvia" };
    printf ("Digite seu nome:");
    gets(nome);
    for (cont=0; cont<5; cont++)
        if (strcmp(list[cont],nome) == 0)
            entra=1;
    if (entra == 1)
        printf ("%s pode ser incluído.", nome);
    else
        printf ("%s não pode ser incluído.", nome);
}
```

# Vetor de strings

## (usando Ponteiros para char)

```
char* list[5];
```

...

	0	1	2	3	4	5	6
list[0] →	C	a	r	l	o	s	\0
list[1] →	A	n	a	\0			
list[2] →	P	e	d	r	o	\0	
list[3] →	A	n	d	r	e	\0	
list[4] →	S	i	l	v	i	a	\0

SEM desperdício  
de Memória

# Exercício

## (Verificando se inclui um nome)

```
main( )
{
    int cont;
    int entra=0;
    char nome[40];
    char *list[5]= {"Carlos","Ana","Pedro","Andre", "Silvia" };
    printf ("Digite seu nome: ");
    gets(nome);
    for (cont=0; cont < 5; cont++)
        if (strcmp(list[cont],nome) == 0)
            entra = 1;
    if (entra == 1)
        printf ("Voce pode entrar");
    else
        printf ("Voce nao pode entrar");
}
```

# Matriz de Caracteres

## Versão matriz

	0	1	2	3	4	5	6
list[0] →	C	a	r	l	o	s	\0
list[1] →	A	n	a	\0			
list[2] →	P	e	d	r	o	\0	
list[3] →	A	n	d	r	e	\0	
list[4] →	S	i	l	v	i	a	\0

Desperdício

## Versão ponteiro

list[0] →	C	a	r	l	o	s	\0
list[1] →	A	n	a	\0			
list[2] →	P	e	d	r	o	\0	
list[3] →	A	n	d	r	e	\0	
list[4] →	S	i	l	v	i	a	\0

# Exercício

## Lista Original

list[0] →	C	a	r	l	o	s	\0
list[1] →	A	n	a	\0			
list[2] →	P	e	d	r	o	\0	
list[3] →	A	n	d	r	e	\0	
list[4] →	S	i	l	v	i	a	\0

## Lista Ordenada

list[0]	↗	A	n	a	\0			
list[1]	↘	A	n	d	r	e	\0	
list[2]	↗	C	a	r	l	o	s	\0
list[3]	↘	P	e	d	r	o	\0	
list[4] →		S	i	l	v	i	a	\0

- Escreva um programa para colocar a lista de nomes apontadas pelo array em ordem alfabética

```

#include "stdio.h"
#include "conio.h"
#include "string.h"
main( )
{
    int cont1, cont2;
    char *temp;
    static char *list[5]={"Carlos","Ana","Pedro","Andre","Silvia"};

    printf("\nLista Original:\n\n");
    for(cont1=0; cont1<5; cont1++)
        printf("Nome %d: %s\n",cont1,list[cont1]);

    for(cont1=0; cont1<5; cont1++)
        for(cont2=cont1+1; cont2 < 5; cont2++)
            if (strcmp(list[cont1] , list[cont2]) > 0){
                temp          = list[cont2];
                list[cont2]   = list[cont1];
                list[cont1]   = temp;
            }
    printf("\nLista Ordenada:\n\n");
    for(cont1=0; cont1 < 5; cont1++)
        printf("Nome %d: %s\n",cont1,list[cont1]);
}

```

# Alocação de Memória

- Em C, podemos controlar a alocação de memória de duas formas:
  - **Alocação estática**
    - Memória alocada criando/declarando variáveis (espaço da memória é definido durante a compilação)
  - **Alocação dinâmica**
    - Memória alocada quando desejarmos e com o espaço que desejarmos

# Função malloc()

- Recebe um inteiro sem sinal como argumento, representando a **quantidade em bytes de memória requerida** (a alocar)
- Retorna um ponteiro para o endereço do primeiro byte de memória que foi alocado



# Exemplo

```
#include <stdlib.h>
main()
{
    char *str;
    /* alocar memoria para uma string */
    str = (char*) malloc (10);
    if (str == NULL)
    {
        printf("Nao ha memoria suficiente\n");
        exit(1); /* terminar programa */
    }
    /* copiar "Hello" na string */
    strcpy(str, "Hello");
    printf("String is %s\n", str);
    free(str);
}
```

# Liberando a Memória

- De um lado, quando **alocamos memória estaticamente** o programa automaticamente libera a memória utilizada pelas variáveis declaradas. Por outro lado, a **alocação dinâmica NÃO libera** a memória automaticamente.
- É importante liberar a memória alocada após o seu uso para evitar **desperdício de espaço**

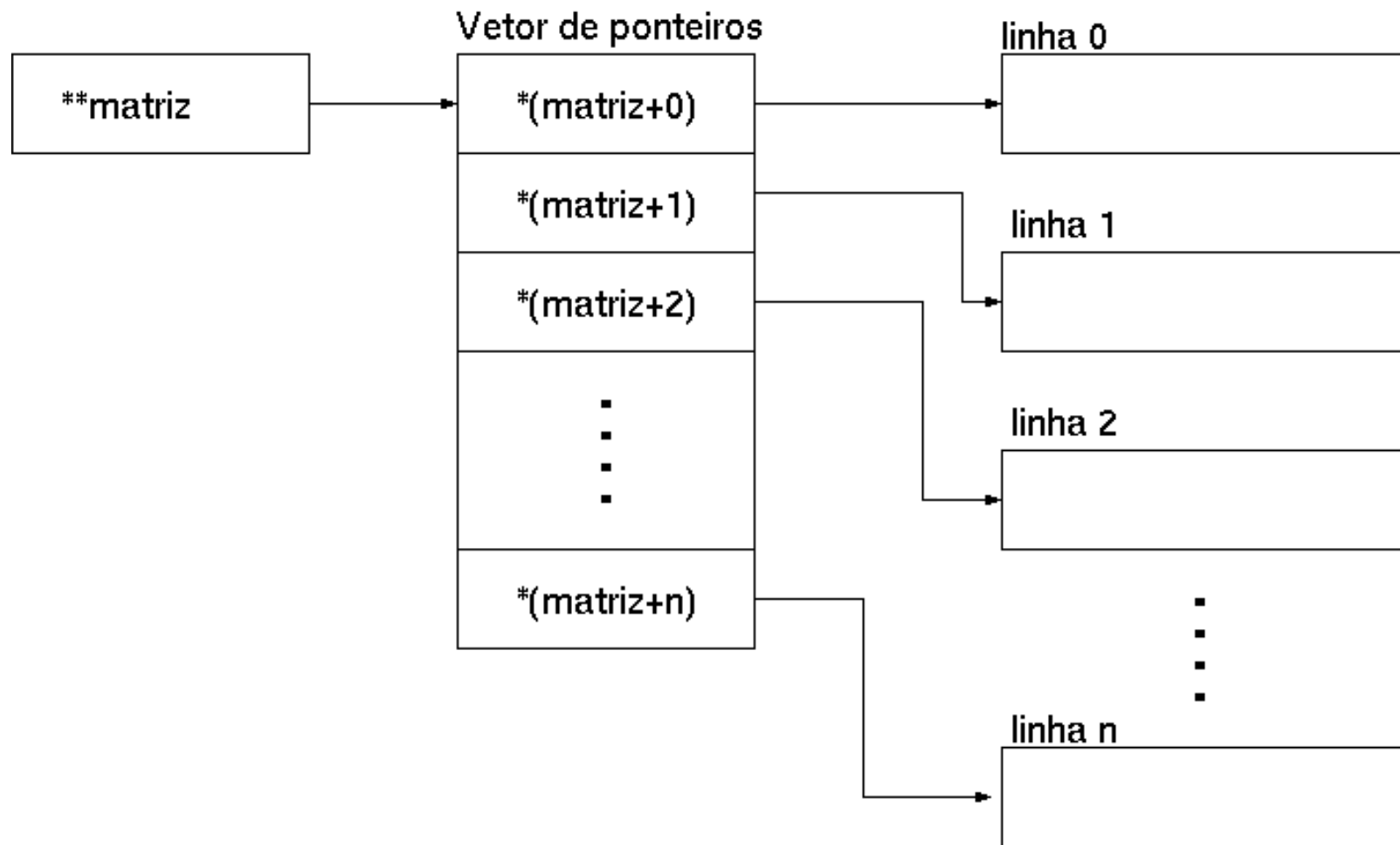
# Função free()

- Complemento de malloc()
- Libera esta área para possível utilização futura
- Argumento: ponteiro para uma área de memória previamente alocada por malloc()

# Alocando Dinâmica e Matrizes

- Deve-se usar um ponteiro para ponteiro
  - `int **matriz`
- É como se você anotasse o endereço/caminho de um arquivo do computador que tem o endereço da casa do seu amigo
  - Para alocar uma matriz com malloc, é preciso fazer a alocação de todas as linhas e, em seguida, dos elementos de cada linha
  - Da mesma forma, a liberação da memória é feita em partes

# Ponteiro para ponteiro



# Alocando Dinamicamente uma Matriz

```
/*alocacao*/  
int **matriz;  
matriz=(int**) malloc(10*sizeof(int*));  
int i,j;  
for (i=0; i<10; i++)  
    matriz[i] = (int*) malloc(5*sizeof(int));  
  
/*inicializacao*/  
for (i=0; i<10; i++)  
    for (j=0; j<5; j++)  
        matriz[i][j] = i+j;  
  
/*impressao*/  
for (i=0; i<10; i++)  
    for (j=0; j<5; j++)  
        printf ("%d\n", matriz[i][j]);  
  
/*liberacao*/  
for (i=0; i<10; i++)  
    free(matriz[i]);  
free(matriz);
```

Alocando  
uma matriz  
10x5 de  
inteiros

# Matrizes como Saída de Funções

- A função abaixo recebe uma matriz e soma um valor qualquer aos elementos dessa matriz e devolve outra matriz de saída....

```
int** add(int **ptr)
{
    int i, j, c = 20;
    for (i=0; i<tam; i++)
        for (j=0; j<tam; j++)
            ptr[i][j]= ptr[i][j]+c;
    return (ptr);
}
```

# Passando Estruturas para Funções

- As novas versões de C permitem que uma função passe como argumento ou produza como retorno uma estrutura completa para outra função



# Exercício

- Criar uma função para obter dos usuários dados sobre os livros

# Passando Estruturas para Funções

## Tipo Global

```
struct livro {  
    char titulo[40];  
    int  regnum;  
};  
typedef struct livro liv;  
main( )  
{  
    liv livro1;  
    liv livro2;  
    livro1 = novonome( );  
    livro2 = novonome( );  
    altera(&livro1);  
    altera(&livro2);  
    list(livro1);  
    list(livro2);  
}
```

```
liv novonome( )  
{  
    char numstr[81];  
    liv livr;  
    printf("\nDigite título:");  
    gets(livr.titulo);  
    printf("Digite registro:");  
    gets(numstr);  
    livr.regnum=atoi(numstr);  
    return(livr);  
}  
void list(liv livr)  
{  
    printf("\nLivro:\n");  
    printf("Título: %s\n", livr.titulo);  
    printf("No do registro: %3d\n",livr.regnum);  
}
```

# Passando Estruturas para Funções

## Tipo Global

```
struct livro {
    char titulo[40];
    int  regnum;
};
typedef struct livro liv;
main( )
{
    liv livro1;
    liv livro2;
    novonome(& livro1 );
    novonome(& livro2 );
    altera(&livro1);
    altera(&livro2);
    list(&livro1);
    list(&livro2);
}
```

```
void novonome(liv* livr )
{
    char numstr[81];
    printf("\nDigite título:");
    gets((*livr).titulo);
    printf("Digite registro:");
    gets(numstr);
    (*livr).regnum=atoi(numstr);
    return(livr);
}
void list(liv* livr)
{
    printf("\nLivro:\n");
    printf("Título: %s\n", (*livr).titulo);
    printf("No do registro: %3d\n",
        (*livr).regnum);
}
```

# Ponteiros para Estruturas

- `struct livro *ptrl;`
  - O ponteiro **ptrl** pode apontar para qualquer estrutura do tipo **livro**
- `ptrl = &livro;`
  - **ptrl** aponta para **livro**

# Ponteiros para Estruturas

Acessando os membros através do ponteiro

- Normalmente, os membros de uma estrutura são acessados usando seu nome seguido do operador ponto
- O mesmo acontece com ponteiros ?
  - Se **struct livro \*ptrl**, poderíamos escrever **ptrl.regnum** ?
  - NÃO !!!

# Ponteiros para Estruturas

Acessando os membros através do ponteiro

- Correto:
  - `ptrl->regnum`
- ou
  - `(*ptrl).regnum`
    - Os parênteses são obrigatórios !!!
    - Sem eles, a estrutura seria lida como `*(ptrl.regnum)` que geraria um erro !

# Ponteiros para Estruturas

```
struct livro {  
    char titulo[30];  
    char autor[30];  
    int regnum;  
    double preco;  
};
```

```
main( )  
{
```

```
    struct livro lista[2] =  
        { { "C", "Carlos", 102, 70.5 },  
          { "C++", "Alexandre", 321, 63.25} };
```

```
    struct livro *ptr1; /* ponteiro para estrutura */
```

```
    printf("Endereço 1: %u    2: %u\n", &lista[0], &lista[1]);
```

```
    ptr1 = &lista[0];
```

```
    printf("Ponteiro 1: %u    2: %u\n", ptr1, ptr1+1);
```

```
    printf("ptr1->preço:R$.%2f (*ptr1).preço: R$.%2f\n",ptr1->preço,  
        (*ptr1).preço);
```

```
    ptr1++; /* aponta para a próxima estrutura */
```

```
    printf("ptr1->título:%s ptr1->autor:%s\n", ptr1->título,ptr1->autor);
```

```
}
```

Endereço 1: 404 2: 476

Ponteiro 1: 404 2: 476

ptr1->preço: R\$ 70.5 (\*ptr1).preço: R\$ 70.5

ptr1->título: C++ ptr1->autor: Alexandre

# Exercício

- Considere que uma empresa precisa armazenar os seguintes dados de um cliente:
  - Nome completo com no máximo 50 caracteres;
  - renda mensal do cliente;
  - ano de nascimento;
  - possui ou não carro.
- Considerando esta estrutura escreva um programa que leia os dados de N clientes e imprima:
  - quantos clientes têm renda mensal acima da média;
  - quantos clientes têm carro;
  - quantos clientes nasceram entre 1960 (inclusive) e 1980 (inclusive).
- Use ponteiros para manipular o vetor de estruturas



# Ponteiros para Funções

```
main (void)
```

```
{
```

```
    char s[20] = "Progrmação C";
```

```
    int (*p)(const char *);
```

```
    p = puts;
```

```
    (*p)(str);
```

```
    p(str);
```

```
    return 0;
```

```
}
```

- COMO FUNCIONA:
- Declaracao do ponteiro para uma função de retorno inteiro e parâmetro uma string constante
- O ponteiro p passa a apontar para a função puts
  - int puts(const char\*)
- Chamada à função puts através do ponteiro p
- Outra forma de se fazer a chamada à função puts usando p

# Passando Funções como Argumento

```
void PrintString(char *str, int (*func)(const char *));
```

```
main (void)
```

```
{  
    char s[20] = "Progrmação C";  
    PrintString (s, puts);  
    return 0;  
}
```

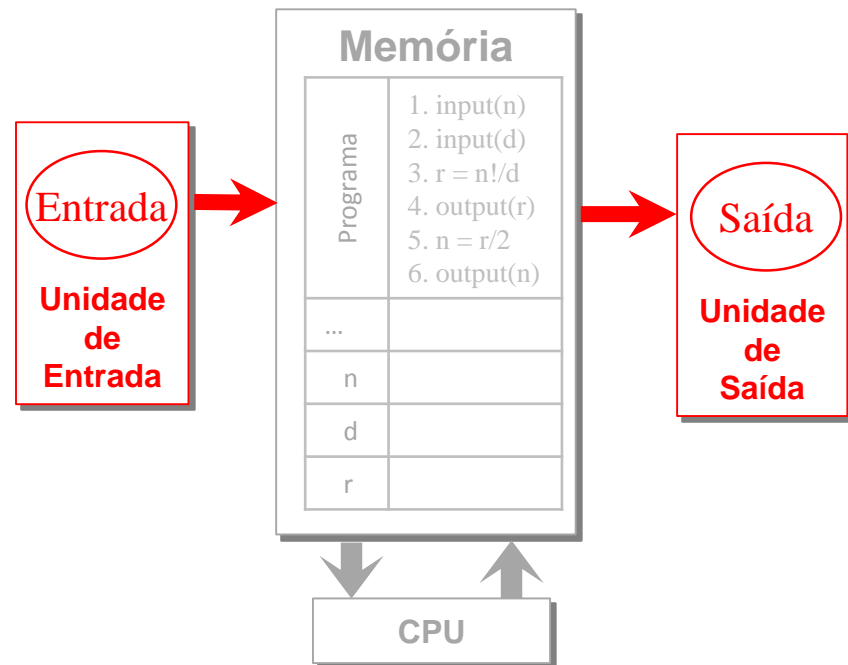
```
void PrintString (char *str, int (*func)(const char *))
```

```
{  
    (*func)(str);  
}
```

- COMO FUNCIONA:
- Função PrintString declara um parâmetro do tipo ponteiro para uma função de retorno inteiro e parâmetro uma string constante
- Função puts é passada como argumento para PrintString
- Chamada à função puts através do ponteiro func (parâmetro de PrintString)

Arquivos

# MÓDULO 10



# Arquivos

- Armazenamento primário – memória
  - Programas em execução e suas variáveis
- Armazenamento secundário – discos
  - Executáveis dos programas
  - Arquivos que armazenam dados
- Programas podem manipular dados de um arquivo
  - Leitura, escrita
  - Disco -> Memória, Memória -> Disco

# Operações com Arquivos em Disco

- Existem dois tipos de arquivo: texto e binário
- Arquivo Texto
  - Seqüência de caracteres agrupados em linhas
  - Linhas separadas por um caractere denominado LF
  - No windows, as linhas são separadas por CR/LF
  - Windows fornece uma indicação de fim de arquivo ao programa quando ele tenta ler alguma informação após último caractere

# Operações com Arquivos em Disco

- Arquivo binário:
  - Nenhuma conversão é feita
    - qualquer caractere é lido ou gravado sem alteração
  - Nenhuma indicação de fim de arquivo é reconhecida
  - dados são armazenados da mesma forma que representados na memória

# Abrindo arquivos

```
FILE *fptr; /* ponteiro para arquivo */
```

```
fptr = fopen("arqtext.txt", "w");
```

nome do arquivo

## Tipo de abertura

- “r”** Abrir arquivo texto para leitura. O arquivo deve estar presente no disco
- “w”** Abrir arquivo texto para gravação. Se o arquivo existir ele será destruído e reinicializado. Se não existir, será criado
- “a”** Abrir um arquivo texto para gravação. Os dados serão adicionados no fim do arquivo existente, ou cria um novo

# Abrindo arquivos

```
FILE *fptr; /* ponteiro para arquivo */  
fptr = fopen("arqtext.txt", "w+");
```

nome do  
arquivo

Tipo de abertura

- “r+”** Abrir arquivo texto para leitura e gravação. O arquivo deve existir e pode ser **atualizado**.
- “w+”** Abrir arquivo texto para leitura e gravação. Se o arquivo existir ele será **destruído e reinicializado**. Se não existir, será criado.
- “a+”** Abrir um arquivo texto para atualização e para adicionar dados no fim do arquivo existente, ou cria um novo



# Abrindo arquivos

```
FILE *fptr; /* ponteiro para arquivo */  
fptr = fopen("arqtext.txt", "wb");
```

nome do arquivo

## Tipo de abertura

- “rb”** Abrir arquivo binário para leitura. O arquivo deve estar presente no disco
- “wb”** Abrir arquivo binário para gravação. Se o arquivo existir ele será destruído e reinicializado. Se não existir, será criado
- “ab”** Abrir um arquivo binário para gravação. Os dados serão adicionados no fim do arquivo existente, ou cria um novo

# Abrindo arquivos

```
FILE *fptr; /* ponteiro para arquivo */  
fptr = fopen("arqtext.txt", "wb+");
```

nome do  
arquivo

## Tipo de abertura

- “rb+”** Abrir arquivo binário para leitura e gravação. O arquivo deve existir e pode ser **atualizado**.
- “wb+”** Abrir arquivo binário para leitura e gravação. Se o arquivo existir ele será **destruído e reinicializado**. Se não existir, será criado.
- “ab+”** Abrir um arquivo binário para atualização e para adicionar dados no fim do arquivo existente, ou cria um novo

# Fechando arquivos

```
FILE *fptr; /* ponteiro para arquivo */  
fptr = fopen("arqtext.txt", "w");  
fclose(fptr);
```

# Escrevendo em arquivos

```
FILE *fptr; /* ponteiro para arquivo */  
char ch;  
fptr = fopen("arqtext.txt", "w");  
while((ch=getche( )) != '\r')  
    fputc(ch, fptr);  
fclose(fptr);
```

# Lendo arquivos

```
FILE *fptr; /* ponteiro para arquivo */  
char ch;  
fptr = fopen("arqtext.txt", "r");  
while((ch=fgetc(fptr)) != EOF)  
    printf("%c",ch);  
fclose(fptr);
```

Fim do arquivo

# Cuidados ao abrir arquivos

- A operação para abertura de arquivos pode falhar !!!
  - Falta de espaço em disco
  - Arquivo ainda não criado
- Se o arquivo não puder ser aberto, a função `fopen( )` retorna o valor `NULL`

# Cuidados ao abrir arquivos

```
FILE *fptr; /* ponteiro para arquivo */
char ch;
if((fptr = fopen("arqtext.txt", "r") == NULL)
{
    printf("Nao foi possivel abrir o arquivo arqtext.txt");
    exit(-1);
}
while((ch=fgetc(fptr)) != EOF)
    printf("%c",ch);
fclose(fptr);
```

# Gravando um arquivo linha a linha

```
FILE *fptr; /* ponteiro para arquivo */
char string[81];
if((fptr = fopen("arqtext.txt", "w") == NULL) {
    printf("Nao foi possivel abrir o arquivo arqtext.txt");
    exit(-1);
}
while(strlen(gets(string)) > 0) {
    fputs(string, fptr);
    fputs("\n", fptr);
}
fclose(fptr);
```



# Lendo um arquivo linha a linha

```
FILE *fptr; /* ponteiro para arquivo */
char string[81];
if((fptr = fopen("arqtext.txt", "r") == NULL) {
    printf("Nao foi possivel abrir o arquivo arqtext.txt");
    exit(-1);
}
while(fgets(string,80,fptr)) != NULL)
    printf("%s",string);
fclose(fptr);
```

## Gravando um arquivo de maneira formatada

É possível usar *fscanf* para ler dados do disco.

*fscanf* é similar à função *scanf*, exceto que, como *fprintf*, um ponteiro para **FILE** é incluído como primeiro argumento

```
FILE *fptr;
char título[30];
int regnum;
float preco;
fptr = fopen("arqtext.txt", "w");
if(fptr== NULL) {
    printf("Nao foi possivel abrir o arquivo arqtext.txt");
    exit(-1);
}
do {
    printf("\nDigite titulo, registro e preco");
    scanf("%s %d %f", título, &regnum, &preco);
    fprintf(fptr,"%s %d %f \n", título, regnum, preco);
} while(strlen(título) > 1) ;
fclose(fptr);
```

## Arquivos Binários: fwrite

```
struct livros {  
    char título[30];  
    int regnum;  
} livro;  
char numstr[81];  
FILE *fptr;  
  
if((fptr = fopen("livros.arq", "wb")) == NULL){...}  
  
do { printf("\n Digite o título:");  
    gets(livro.título);  
    printf("\n Digite o registro:");  
    scanf("%d",&livro.regnum);  
    fwrite(&livro, sizeof(struct livros), 1, fptr);  
    printf("\n Adiciona outro livro (s/n) ?");  
} while (getche( ) == 's');  
fclose(fptr);
```

# Gravando vetores com fwrite

```
int tabela[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
FILE *fptr;
```

```
if((fptr = fopen("tabela.arq", "wb")) == NULL)  
{...}
```

```
fwrite(tabela, sizeof(int), 10, fptr);
```

```
fclose(fptr);
```

# Lendo Estruturas com fread

```
struct livro livro;  
char numstr[81];  
FILE *fptr;  
if((fptr = fopen("livros.arq", "rb")) == NULL) {...}  
  
while (fread(&livro, sizeof(struct livros), 1, fptr) == 1) {  
    printf("\n Título: %s\n", livro.título);  
    printf("\n Registro: %03d\n", livro.regnum);  
}  
fclose(fptr);
```

# Navegação em arquivos

- `rewind(FILE * p)`
  - Reposiciona o indicador de posição no início do arquivo
  - Arquivo precisa ser aberto no modo leitura/escrita (+), se precisar ler e escrever ao mesmo tempo
- `fseek(FILE *p, long numbytes, int origem)`
  - Modifica o indicador de posição para qualquer localidade dentro do arquivo
  - **origem** usa uma das seguintes constantes (`stdio.h`)
    - `SEEK_SET`: início do arquivo
    - `SEEK_CUR`: posição atual
    - `SEEK_END`: final do arquivo

# Navegação em arquivos

```
char str[80];  
FILE *fp;  
  
if ((fp = fopen("teste.txt", "w+")) == NULL) { .. }  
  
gets(str);  
strcat(str, "\n"); //acrescenta nova linha  
fputs(str, fp);  
/* agora le a string */  
rewind(fp);  
fgets(str, 79, fp);  
printf(str);  
  
fclose(fp);
```

# Navegação em arquivos

```
struct livro livro;  
FILE *fp;  
if ((fp = fopen("teste.dat", "rb")) == NULL) { .. }  
  
/* procura décima estrutura no arquivo binario */  
int i = fseek(fp, 9*sizeof(struct livro), SEEK_SET);  
if (i) {  
    /*devolve diferente de zero se houver erro*/  
}  
fread(&livro, sizeof(struct livros), 1, fp);  
  
fclose(fp);
```



# Exercício

- Escreva um programa que grava um texto qualquer em um arquivo e, em seguida, conta o número de caracteres do arquivo
  - Use a função `rewind(FILE * p)` para “rebobinar” o arquivo
  - Observe o tamanho do arquivo e o número de caracteres contados (usar comando `dir` do DOS)

# Exercício

- Escreva um programa que grave o nome, 1a nota, 2a nota e média dos alunos de uma turma em um arquivo. Após a gravação do arquivo, ele deve ser novamente aberto para listar os alunos com suas respectivas notas.

# Engenharia da Computação

[www.eComp.Poli.br](http://www.eComp.Poli.br)

## Linguagem C

Linguagem de Programação Imperativa (LPI)

Prof. Joabe Jesus

[joabe@ecomp.poli.br](mailto:joabe@ecomp.poli.br)