

Linguagem C

Histórico

- Desenvolvida por **Dennis Ritchie** na década de 1970 num laboratório da AT&T.
- Tinha o propósito de ser uma linguagem de alto nível com facilidades de baixo nível.
- Com a popularidade conseguida nos anos 1980, começaram a surgir diversos compiladores com características diferentes.
 - Algo indesejável para uma linguagem de alto nível, que normalmente almeja portabilidade.

Histórico

Há quem diga que C é uma linguagem de *nível médio* por isso.

- Desenvolvida por **Dennis Ritchie** na década de 1970 num laboratório da AT&T.
- Tinha o propósito de ser uma linguagem de alto nível com facilidades de baixo nível.
- Com a popularidade conseguida nos anos 1980, começaram a surgir diversos compiladores com características diferentes.
 - Algo indesejável para uma linguagem de alto nível, que normalmente almeja **portabilidade**.

Queremos que nosso **código compile na maioria das plataformas** sem alterações. Quando compiladores apresentam características diferentes, eles criam *dialetos* do C que não são compatíveis entre si.

Histórico

- Para unificar a linguagem (e compiladores) foi necessário padronizar:
 - C89 -> C90 (ANSI C)
 - C99
 - C11
- Entretanto, mesmo com padrões, algumas construções podem ter interpretações diferentes em compiladores diferentes.
 - Devido ambiguidades ou omissões.
 - Não foram corrigidas pelos novos padrões por compatibilidade com programas antigos.
 - São as construções ***dependentes de implementação*** [do compilador].

Histórico

Primeiro padrão e ainda muito utilizado. Sempre que possível é bom respeitar esse padrão.

- Para unificar a linguagem (e compiladores) foi necessário padronizar:

- C89 -> C90 (ANSI C)
- C99
- C11

Já existiam muitos programas antes da padronização e alguns poderiam deixar de funcionar como esperado.

- Entretanto, mesmo com padrões, algumas construções podem ter interpretações diferentes em compiladores diferentes.

- Devido ambiguidades ou omissões.
- Não foram corrigidas pelos novos padrões por compatibilidade com programas antigos.
- São as construções ***dependentes de implementação*** [do compilador].

São construções/expressões que podem dar resultados diferentes de acordo com o compilador utilizado. Devemos conhecê-las para evitar o uso.

Identificadores

- Um **identificador** serve para **dar nome** aos componentes utilizados nos programas escritos em linguagens de programação.
- Existem **regras** definidas para identificadores em cada linguagem.

Identificadores

Por exemplo, dar nome as variáveis. O nome da variável é seu identificador.

- Um **identificador** serve para **dar nome** aos componentes utilizados nos programas escritos em linguagens de programação.
- Existem **regras** definidas para identificadores em cada linguagem.

Identificadores em C

- As regras para um identificador válido em C são:
 - Ser composto apenas por letras, dígitos e sublinha (“_”).
 - O primeiro caractere não pode ser dígito (*número*).
 - Apesar de não existir tamanho máximo definido, o padrão garante pelo menos 31 caracteres válidos.
 - Os padrões mais novos aumentaram esse limite para 63.
 - Lembrar, no entanto, que identificadores são nomes e não frases ou parágrafos.
- Atenção: C faz distinção entre letras maiúsculas e minúsculas!

Identificadores em C

- As regras para um identificador válido em C são:
 - Ser composto apenas por letras, dígitos e sublinha (“_”).
 - O primeiro caractere não pode ser dígito (*número*).
 - Apesar de não existir tamanho máximo definido, o padrão garante pelo menos 31 caracteres válidos.
 - Os padrões mais novos aumentaram esse limite para 63.
 - Lembrar, no entanto, que identificadores são nomes e não frases ou parágrafos.
- Atenção: C faz distinção entre letras maiúsculas e minúsculas!

Por exemplo, *media* e *Media* são variáveis diferentes — e seu uso diminuiria a **qualidade** do código-fonte.

Identificadores em C

- As ***palavras-chaves*** da linguagem **não podem** ser utilizadas para criação de identificadores.
 - São as palavras que possuem significado especial na linguagem, como ***if***, ***while***, ***else***, etc.
- As ***palavras reservadas*** são identificadores disponibilizados pela biblioteca padrão do C (como o ***printf*** e o ***scanf***) e o programadores **devem evitar** redefini-los para não os inutilizar.
- Identificadores iniciados por sublinha também são reservados pela linguagem e devem ser evitados.

Códigos de Caracteres

- A linguagem define um conjunto básico de caracteres, com um subconjunto mínimo de 96 bem definidos (os mais comuns).
 - Cada **caractere** deve caber em ***um byte*** (o tamanho ocupado na memória).
 - Cada caractere possui um número inteiro a ele atribuído.
- Um **código de caracteres** consiste no **mapeamento** entre um conjunto de caracteres e um conjunto de números inteiros. Exemplo:
 - ASCII
 - EBDIC
- **Nunca** em um programa suponha qual código seja utilizado na execução.

Códigos de Caracteres

- A linguagem define um conjunto básico de caracteres, com um subconjunto mínimo de 96 bem definidos (os mais comuns).
 - Cada **caractere** deve caber em **um byte** (o tamanho ocupado na memória).
 - Cada caractere possui um número inteiro a ele atribuído.
- Um **código de caracteres** consiste no **mapeamento** entre um conjunto de caracteres e um conjunto de números inteiros. Exemplo:
 - ASCII
 - EBDIC
- **Nunca** em um programa suponha qual código seja utilizado na execução.

Mas, até hoje só vi **ASCII**.

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(72	48	110	H	104	68	150	h
9	9	11		41	29	51)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	
29	1D	35		61	3D	75	=	93	5D	135]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

ASCII Table

Esses caracteres não possuem representação gráfica. O **ENTER** (`'\n'`) é o **13**, por exemplo.

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107		103	67	147	g
8	8	10		40	28	50	(72	48	110	H	104	68	150	h
9	9	11		41	29	51)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	70	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	71	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	72	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	73	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	74	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	75	157	o
16	10	20		48	30	60	0	80	50	120	P	112	76	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	77	161	q
18	12	22		50	32	62	2	82	52	122	R	114	78	162	r
19	13	23		51	33	63	3	83	53	123	S	115	79	163	s
20	14	24		52	34	64	4	84	54	124	T	116	80	164	t
21	15	25		53	35	65	5	85	55	125	U	117	81	165	u
22	16	26		54	36	66	6	86	56	126	V	118	82	166	v
23	17	27		55	37	67	7	87	57	127	W	119	83	167	w
24	18	30		56	38	70	8	88	58	130	X	120	84	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	85	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	86	172	z
27	1B	33		59	3B	73	;	91	5B	133	[123	87	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	88	174	
29	1D	35		61	3D	75	=	93	5D	135	^	125	89	175	}
30	1E	36		62	3E	76	>	94	5E	136	_	126	90	176	~
31	1F	37		63	3F	77	?	95	5F	137		127	91	177	

Quando uma variável do tipo **char** armazena o valor 'a' e o compilador utiliza a tabela **ASCII**, na memória essa variável na verdade armazena o valor inteiro **97**.

Os valores inteiros dos dígitos são diferentes dos números que representam.

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(72	48	110	H	104	68	150	h
9	9	11		41	29	51)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[123	7B	173	[
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	\
29	1D	35		61	3D	75	=	93	5D	135	^	125	7D	175	^
30	1E	36		62	3E	76	>	94	5E	136	_	126	7E	176	_
31	1F	37		63	3F	77	?	95	5F	137	`	127	7F	177	`

Não devemos decorar a tabela. No entanto, é importante saber que o alfabeto aparece em ordem, assim como os dígitos e que as letras maiúsculas vêm antes que as minúsculas — portanto, possuem códigos menores.

Tipos de Dados Primitivos

- Um **tipo de dado** consiste de um conjunto de valores munidos de uma coleção de operações permitidas sobre eles,
- Os **tipos primitivos** são os embutidos na linguagem,
 - É possível criar novos tipos, os **tipos derivados**.
- Existem diversos tipos primitivos em C, mas vamos trabalhar com apenas alguns na disciplina.

Tipos de Dados

Por exemplo, o tipo **int** permite números inteiros e operações como soma, multiplicação, divisão, resto da divisão inteira e outras. Já o tipo **float** não permite o resto da divisão inteira, por exemplo.

- Um **tipo de dado** consiste de um conjunto de valores munidos de uma coleção de operações permitidas sobre eles,
- Os **tipos primitivos** são os embutidos na linguagem,
 - É possível criar novos tipos, os **tipos derivados**.
- Existem diversos tipos primitivos em C, mas vamos trabalhar com apenas alguns na disciplina.

Tipos de Dados Primitivos

- ***int***

- Largura não definida! Um dos defeitos dos primórdios da linguagem.
- Usar ***<stdint.h>*** em programas sérios.

- ***char***

- Ocupa sempre **um byte**.
- Pode ser usado para inteiros que requerem apenas um byte.
 - talvez armazene um número inteiro não associado a um caractere.

- ***double*** (dupla precisão) e ***float*** (precisão simples)

- Não consegue representar muito bem todos os números reais.
 - Como qualquer outro tipo de ponto flutuante.

Tipos de Dados

Primeira característica *dependente de implementação* que conheceremos. Em alguns compiladores um ***int*** pode ocupar **2 bytes** na memória, em outros **4** ou **8**. O padrão da linguagem não define uma largura única.

- ***int***

- Largura não definida! Um dos defeitos dos primórdios da linguagem.
- Usar ***<stdint.h>*** em programas sérios.

- ***char***

- Ocupa sempre **um byte**.
- Pode ser usado para inteiros que requerem apenas um byte.
 - talvez armazene um número inteiro não associado a um caractere.

Independente do compilador, um ***char*** terá sempre **1 byte** — 256 valores diferentes.

- ***double*** (dupla precisão) e ***float*** (precisão simples)

- Não consegue representar muito bem todos os números reais.
 - Como qualquer outro tipo de ponto flutuante.

Tipos de Dados Primitivos

- ***int***

- Largura não definida! Um dos defeitos dos primórdios da linguagem.
- Usar ***<stdint.h>*** em programas sérios.

Se você tiver uma variável para, por exemplo, armazenar uma *idade*, você pode usar o tipo ***char*** como um *inteiro pequeno*.

- ***char***

- Ocupa sempre **um byte**.
- Pode ser usado para inteiros que requerem apenas um byte.
 - talvez armazene um número inteiro não associado a um caractere.

- ***double*** (dupla precisão) e ***float*** (precisão simples)

- Não consegue representar muito bem todos os números reais.
 - Como qualquer outro tipo de ponto flutuante.

Tipo	Bytes	Escala
char	1	-128 a 127
int	4	-2.147.483.648 a 2.147.483.647
short	2	-32.765 a 32.767
long	4	-2.147.483.648 a 2.147.483.647
unsigned char	1	0 a 255
unsigned	4	0 a 4.294.967.295
unsigned long	4	0 a 4.294.967.295
unsigned short	2	0 a 65.535
float	4	$3,4 \times 10^{-38}$ a $3,4 \times 10^{38}$
double	8	$1,7 \times 10^{-308}$ a $3,4 \times 10^{308}$
long double	10	$3,4 \times 10^{-4932}$ a $3,4 \times 10^{4932}$
void	0	nenhum valor

C99 define também o long long com 8bytes.

Tipo	Bytes	Escala
char	1	-128 a 127
int	4	-2.147.483.648 a 2.147.483.647
short	2	-32.765 a 32.767
long	4	-2.147.483.648 a 2.147.483.647
unsigned char	1	0 a 255
unsigned	4	0 a 4.294.967.295
unsigned long	4	0 a 4.294.967.295
unsigned short	2	0 a 65.535
float	4	$3,4 \times 10^{-38}$ a $3,4 \times 10^{38}$
double	8	$1,7 \times 10^{-308}$ a $3,4 \times 10^{308}$
long double	10	$3,4 \times 10^{-4932}$ a $3,4 \times 10^{4932}$
void	0	nenhum valor

Largura.

Os valores que as variáveis podem assumir/armazenar.

C99 define também o long long com 8bytes.

Tipo	Bytes	Escala
char	1	-128 a 127
int	4	-2.147.483.648 a 2.147.483.647
short	2	-32.765 a 32.767
long	4	-2.147.483.648 a 2.147.483.647
unsigned char	1	0 a 255
unsigned	4	0 a 4.294.967.295
unsigned long	4	0 a 4.294.967.295
unsigned short	2	0 a 65.535
float	4	3,4 x 10 ⁻³⁸ a 3,4 x 10 ³⁸
double	8	1,7 x 10 ⁻³⁰⁸ a 1,7 x 10 ³⁰⁸
long double	10	3,4 x 10 ⁻⁴⁹³² a 3,4 x 10 ⁴⁹³²
void	0	nenhuma

C99 define também o long long com 8bytes.

Os qualificadores não criam novos tipos, apenas modificam os existentes. **short** e **long** diminuem ou aumentam a largura (e a escala). **unsigned** e **signed** especificam que apenas valores sem ou com sinal serão utilizados. Quando o tipo é **int** podemos usar apenas o qualificador, como mostrado na tabela.

Tipo	Bytes	Escala
char	1	-128 a 127
int	4	-2.147.483.648 a 2.147.483.647
short	2	-32.765 a 32.767
long	4	-2.147.483.648 a 2.147.483.647
unsigned char	1	0 a 255
unsigned	4	0 a 4.294.967.295
unsigned long	4	0 a 4.294.967.295
unsigned short	2	0 a 65.535
float	4	$3,4 \times 10^{-38}$ a $3,4 \times 10^{38}$
double	8	$1,7 \times 10^{-308}$ a $3,4 \times 10^{308}$
long double	10	$3,4 \times 10^{-4932}$ a $3,4 \times 10^{4932}$
void	0	nenhum valor

[vídeo](#) com exemplos.

C99 define também o long long com 8bytes.

Constantes

- Existem 5 tipos de constantes em C:
 - inteiras;
 - reais;
 - caracteres;
 - *strings* e de
 - enumerações.
- As constantes são os valores que usamos em expressões e instruções no código-fonte.

Constantes

- Inteiras

- Base decimal

- Ex: **1**, **20** e **-5**

- Base octal

- Ex: **01**, **07** e **010**

- Base hexadecimal

- Ex: **0x10**, **0x1A** e **0X5**

DECIMAL (BASE 10)	BINARY (BASE 2)	OCTAL (BASE 8)	HEXADECIMAL (BASE 16)
0	00000	0	0
1	00001	1	1
2	00010	2	2
3	00011	3	3
4	00100	4	4
5	00101	5	5
6	00110	6	6
7	00111	7	7
8	01000	10	8
9	01001	11	9
10	01010	12	A
11	01011	13	B
12	01100	14	C
13	01101	15	D
14	01110	16	E
15	01111	17	F
16	10000	20	10

Constantes

- Ponto flutuante
 - Formato decimal com ponto
 - Exemplos: **3.1415**, **.5** e **7**.
 - Em notação científica
 - Exemplos: **2E4**, **2e4**, **2.5E-3** e **-3e4**

Constantes

- Caractere

- Exemplos: **'a'**, **'d'**, **' '** (espaço), **'!'**, **'#'**, **'\n'**, **'\t'** e **'\\'**

SEQÜÊNCIA DE ESCAPE	DESCRIÇÃO	FREQÜÊNCIA DE USO
\a	Campainha (alerta)	Raramente usada ¹³
\t	Tabulação	Freqüentemente usada ¹⁴
\n	Quebra de linha	Muito usada ¹⁵
\r	Retorno	Raramente usada ¹⁶
\0	Caractere nulo	Muito usada ¹⁷
\\	Barra invertida	Eventualmente usada
\?	Interrogação	Raramente necessária
\'	Apóstrofo	Eventualmente usada

Constantes

- *Strings*
 - Exemplos: *"oi"*, *"pos-doc na Alemanha"*, *"Dia\tMes\tAno\n"*, *"2020"* e *"?"*

Constantes

- Existem 5 tipos de constantes em C:
 - inteiras;
 - reais;
 - caracteres;
 - *strings* e de
 - enumerações.
- As constantes são os valores que usamos em expressões e instruções no código-fonte.

[Vídeo](#) com exemplos.

Constantes simbólicas

- A linguagem C permite associar identificadores a um valor constante.
 - Esse identificador é uma **constante simbólica**.

```
#define PI 3.14  
#define MAX_ALUNOS 60  
#define SEGREDO 5
```

- As principais vantagens do uso de constantes simbólicas são:
 - Deixar o programa mais **legível**; e
 - Tornar o programa mais fácil de ser modificado (melhor **manutenibilidade**).

Constantes simbólicas

- A linguagem C permite associar identificadores a um valor constante.
 - Esse identificador é uma **constante simbólica**.

```
#define PI 3.14  
#define MAX_ALUNOS 60  
#define SEGREDO 5
```

Por convenção, utilizamos apenas letras maiúsculas nos identificadores das constantes simbólicas.

Perceba que não usamos ;

- As principais vantagens do uso de constantes simbólicas são:
 - Deixar o programa mais **legível**; e
 - Tornar o programa mais fácil de ser modificado (melhor **manutenibilidade**).

Constantes simbólicas

Antes da compilação, todas as ocorrências dos identificadores no código serão substituídas pelo valor associado, como se fosse um *procurar/substituir*.

- A linguagem C permite associar identificadores a um valor constante.
 - Esse identificador é uma **constante simbólica**.

```
#define PI 3.14  
#define MAX_ALUNOS 60  
#define SEGREDO 5
```

- As principais vantagens do uso de constantes simbólicas são:
 - Deixar o programa mais **legível**; e
 - Tornar o programa mais fácil de ser modificado (melhor **manutenibilidade**).

Constantes simbólicas

Antes da compilação, todas as ocorrências dos identificadores no código serão substituídas pelo valor associado, como se fosse um *procurar/substituir*.

- A linguagem C permite associar identificadores a um valor constante.
 - Esse identificador é uma **constante simbólica**.

Isso torna a atualização mais simples, pois só precisamos alterar a constante simbólica uma vez e todas as ocorrências no código serão atualizadas.

```
#define PI 3.14159
#define MAX_ALUNOS 30
#define SEGREDO 7
```

- As principais vantagens do uso de constantes simbólicas são:
 - Deixar o programa mais **legível**; e
 - Tornar o programa mais fácil de ser modificado (melhor **manutenibilidade**).

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

```
int main(void){  
int palpite;
```

```
puts("Adivinhe o numero:");  
scanf("%d", &palpite);
```

```
while(palpite != 5){  
    if (palpite > 5){  
        puts("Palpite alto!");  
    }else{  
        puts("Palpite foi baixo!");  
    }  
}
```

```
    printf("Digite seu novo palpite: ");  
    scanf("%d", &palpite);
```

```
    }  
    puts("Voce ganhou!");
```

```
return 0;  
}
```

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

```
int main(void){  
int palpite;
```

```
puts("Adivinhe o numero:");  
scanf("%d", &palpite);
```

```
while(palpite != 5){  
    if (palpite > 5){  
        puts("Palpite alto!");  
    }else{  
        puts("Palpite foi baixo!");  
    }  
}
```

```
printf("Digite seu novo palpite: ");  
scanf("%d", &palpite);
```

```
}  
puts("Voce ganhou!");
```

```
return 0;  
}
```

Perceba como fica mais legível e dá um significado ao valor 5.

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

```
#define SEGREDO 5
```

```
int main(void){  
int palpite;
```

```
puts("Adivinhe o numero:");  
scanf("%d", &palpite);
```

```
while(palpite != SEGREDO){  
    if (palpite > SEGREDO){  
        puts("Palpite alto!");  
    }else{  
        puts("Palpite foi baixo!");  
    }  
}
```

```
printf("Digite seu novo palpite: ");  
scanf("%d", &palpite);
```

```
}  
puts("Voce ganhou!");
```

```
return 0;  
}
```

Propriedades dos Operadores

- C é uma linguagem intensivamente baseada no uso de operadores.
- Propriedades que todos os operadores do C possuem:
 - Resultado
 - Aridade
 - Precedência
 - Associatividade
- Propriedades que alguns operadores do C possuem:
 - Efeito colateral
 - Ordem de avaliação
 - Curto circuito

Propriedades dos Operadores

- C é uma linguagem intensivamente baseada no uso de operadores.
- Propriedades que todos os operadores do C possuem:
 - Resultado
 - Aridade
 - Precedência
 - Associatividade
- Propriedades que alguns operadores do C possuem:
 - Efeito colateral
 - Ordem de avaliação
 - Curto circuito

O uso altera o valor de alguma variável.

Geralmente tais operadores são utilizados justamente pelo efeito colateral.

Propriedades dos Operadores

- C é uma linguagem intensivamente baseada no uso de operadores.
- Propriedades que todos os operadores do C possuem:
 - Resultado
 - Aridade
 - Precedência
 - Associatividade
- Propriedades que alguns operadores do C possuem:
 - Efeito colateral
 - Ordem de avaliação
 - Curto circuito

Por exemplo, ao somar $x + y$, pode ser feito:

$$\begin{array}{r} x \\ + y \\ \hline z \end{array} \quad \text{ou} \quad \begin{array}{r} y \\ + x \\ \hline z \end{array}$$

Qual dos operandos é avaliado primeiro?

Propriedades dos Operadores

- C é uma linguagem intensivamente baseada no uso de operadores.
- Propriedades que todos os operadores do C possuem:
 - Resultado
 - Aridade
 - Precedência
 - Associatividade
- Propriedades que alguns operadores do C possuem:
 - Efeito colateral
 - Ordem de avaliação
 - Curto circuito

Por exemplo, ao somar $x + y$, pode ser feito:


$$\begin{array}{r} x \\ + y \\ \hline z \end{array} \quad \text{ou} \quad \begin{array}{r} y \\ + x \\ \hline z \end{array}$$

Qual dos operandos é avaliado primeiro?

Propriedades dos Operadores

- C é uma linguagem intensivamente baseada no uso de operadores.
- Propriedades que todos os operadores do C possuem:
 - Resultado
 - Aridade
 - Precedência
 - Associatividade
- Propriedades que alguns operadores do C possuem:
 - Efeito colateral
 - Ordem de avaliação
 - Curto circuito

Algumas vezes apenas um dos operandos será avaliado.



Expressões e Operadores

OPERADOR	OPERAÇÃO
-	Menos unário (inversão de sinal)
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão inteira

x	!x
0	1
diferente de 0	0

x	y	x && y	x y
0	0	0	0
0	diferente de 0	0	1
diferente de 0	0	0	1
diferente de 0	diferente de 0	1	1

Operador	Denominação	Aplicação	Resultado
>	maior do que	a > b	1 se a é maior do b; 0, caso contrário
>=	maior do que ou igual a	a >= b	1 se a é maior do que ou igual a b; 0, caso contrário
<	menor do que	a < b	1 se a é menor do b; 0, caso contrário
<=	menor do que ou igual a	a <= b	1 se a é menor do que ou igual a b; 0, caso contrário
==	igual a	a == b	1 se a é igual a b; 0, caso contrário
!=	diferente de	a != b	1 se a é diferente de b; 0, caso contrário

Expressões e Operadores

GRUPO DE OPERADORES	PRECEDÊNCIA
!, - (unários)	Mais alta
*, /, %	↓
+, - (binários)	↓
>, >=, <, <=	↓
==, !=	↓
&&	↓
	Mais baixa

Definição de Variáveis

- Definir uma variável objetiva:
 - Dar uma interpretação para um espaço de memória (variável).
 - Alocar espaço suficiente para essa variável.
 - Associar esse espaço a um identificador.
- Declarar variável é diferente
 - é apenas dizer que em algum local do código uma variável foi definida.
 - é apenas uma alusão.

Definição de Variáveis

```
int variavel;  
double outraVariavel, umaOutraVariavel;
```

Definição de Variáveis

```
int variavel;  
double outraVariavel, umaOutraVariavel;
```

De uma maneira geral, ao serem definidas as variáveis possuem um valor indefinido, comumente chamado *lixo de memória*. Por isso nunca devemos utilizar esse valor antes de fazer uma atribuição.

Definição de Variáveis

```
int variavel;  
double outraVariavel, umaOutraVariavel;
```

De uma maneira geral, ao serem definidas as variáveis possuem um valor indefinido, ~~comumente~~ chamado *lixo de memória*. Por isso nunca devemos utilizar esse valor antes de fazer uma atribuição.

Esse lixo nada mais é do que os *bits* deixados no espaço da memória em que a variável foi alocada.

Definição de Variáveis

```
int variavel = 0;  
double outraVariavel = 1.5, umaOutraVariavel;
```


Definição de Variáveis

Ao fazer uma *iniciação*, indicamos o valor inicial da variável na definição.

```
int variavel = 0;  
double outraVariavel = 1.5, umaOutraVariavel;
```

Operador de Atribuição

Uma atribuição é na verdade a aplicação do operador de atribuição: =

- Efeito colateral
 - Altera o valor da variável (sempre do lado esquerdo) para o valor da expressão (sempre do lado direito).
- Resultado
 - O valor atribuído à variável.
- Múltiplas atribuições
 - Devido associatividade à direita, é possível fazer múltiplas atribuições.
- Cuidado com a ordem de avaliação dos demais operadores.
- Operadores de atribuições aritméticas.

Operador de Atribuição

- **Efeito colateral**
 - Altera o valor da variável (sempre do lado esquerdo) para o valor da expressão (sempre do lado direito).

- **Resultado**
 - O valor atribuído à variável.

O **resultado** do operador `=` na (e da) expressão `x = 4 + 2` é **6** e o **efeito colateral** é o de alterar o valor de `x` para **6**.

- Múltiplas atribuições
 - Devido associatividade à direita, é possível fazer múltiplas atribuições.
- Cuidado com a ordem de avaliação dos demais operadores.
- Operadores de atribuições aritméticas.

Operador de Atribuição

- Efeito colateral
 - Altera o valor da variável (sempre do lado esquerdo) para o valor da expressão (sempre do lado direito).
- Resultado
 - O valor atribuído à variável.
- Múltiplas atribuições
 - Devido associatividade à direita, é possível fazer múltiplas atribuições.
- Cuidado com a ordem de avaliação dos demais operadores.
- Operadores de atribuições aritméticas.

```
outraVariavel = umaOutraVariavel = 5.2;
```

Operador de Atribuição

- Efeito colateral
 - Altera o valor da variável (sempre do lado esquerdo) para o valor da expressão (sempre do lado direito).
- Resultado
 - O valor atribuído à variável.
- Múltiplas atribuições
 - Devido associatividade à direita, é possível fazer múltiplas atribuições.
- Cuidado com a ordem de avaliação dos demais operadores.
- Operadores de atribuições aritméticas.

`+=, -=, *=, /=`

Operador de Atribuição

- Efeito colateral
 - Altera o valor da variável (sempre do lado esquerdo) para o valor da expressão (sempre do lado direito).
- Resultado
 - O valor atribuído à variável.
- Múltiplas atribuições
 - Devido associatividade à direita, é possível fazer múltiplas atribuições.
- Cuidado com a ordem de avaliação dos demais operadores.
- Operadores de atribuições aritméticas.

[Vídeo](#) com exemplos.

Curto Circuito

- Operador &&
 - Quando o **primeiro operando** tem valor **zero**, o segundo operando não é avaliado.
 - E o **resultado** do operador é **zero**.
- Operador ||
 - Quando o **primeiro operando** é **diferente de zero**, o segundo operando não é avaliado.
 - E o resultado do **operador** é **um**.
- Os dois operadores possuem **ordem de avaliação** especificada como sendo da **esquerda para a direita**.

Curto Circuito

- Operador &&
 - Quando o **primeiro operando** tem valor **zero**, o segundo operando não é avaliado.
 - E o **resultado** do operador é **zero**.
- Operador ||
 - Quando o **primeiro operando** é **diferente de zero**, o segundo operando não é avaliado.
 - E o resultado do **operador** é **um**.
- Os dois operadores possuem **ordem de avaliação** especificada como sendo da **esquerda para a direita**.

[Vídeo](#) com exemplos.

Conversões de Tipos

- A linguagem permite misturar operandos de tipos [aritméticos] diferentes em uma expressão.
 - Por exemplo: $1 + 2.5$
- No entanto, para que um operador seja aplicado, seus operandos devem ser do mesmo tipo.
- Então, conversões de tipos devem ser realizadas.
 - Implícita ou explicitamente .

Conversões Implícitas (ou Automáticas)

- Convertem operandos para um mesmo tipo
 - Podem causar resultados inesperados
 - Programador deve entender como elas ocorrem para evitar problemas.
 - Evitar (e muito!) misturar tipos com e sem sinal.

Conversões Implícitas (ou Automáticas)

- **Conversão de Atribuição**

- O valor do lado direito da expressão é convertido para o tipo da variável do lado esquerdo.

- **Conversão Aritmética Usual**

- Quando um operador possui operandos de tipos diferentes, um deles é convertido no tipo de outro, para que sejam do mesmo tipo.
 - Quando temos operando inteiro (***int*** ou ***char***) e outro real (***double***), o inteiro é convertido para real.
 - Quando um operando é ***int*** e o outro char, o char é convertido para ***int***.

Conversões Implícitas (ou Automáticas)

- **Conversão de Atribuição**

```
char c;  
int variavel;
```

```
c = 835;
```

```
variavel = 184E100;
```

```
variavel = 1.5;
```

- **Conversão Aritmética Usual**

```
int variavel;  
double outraVariavel;
```

```
variavel = 'A' + 5;
```

```
outraVariavel = 1 + 2.5;
```

```
outraVariavel = variavel - 1.2;
```

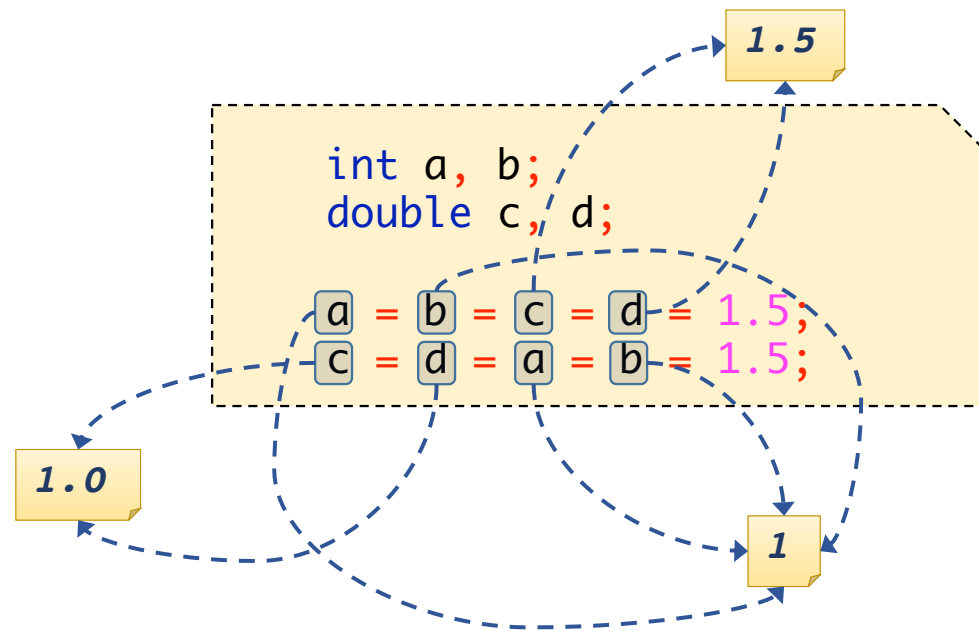
Conversões Implícitas (ou Automáticas)

- Quais os valores atribuídos às variáveis?

```
int a, b;  
double c, d;  
  
a = b = c = d = 1.5;  
c = d = a = b = 1.5;
```

Conversões Implícitas (ou Automáticas)

- Quais os valores atribuídos às variáveis?



[Vídeo](#) com o exemplo.

```
#include <stdio.h>
int main(void)
{
    double preco = 9.90,
           pago = 10.0,
           troco,
           diferenca;
    int    reais,
           centavos;

    troco = pago - preco;
    printf( "\npreco = %f\npago = %f\ntroco = %f\n", preco, pago, troco );

    reais = troco;
    diferenca = troco - reais;
    printf("\ndiferenca = %f\n", diferenca);

    centavos = diferenca*100;
    printf("\nreais = %d\ncentavos = %d\n", reais, centavos);

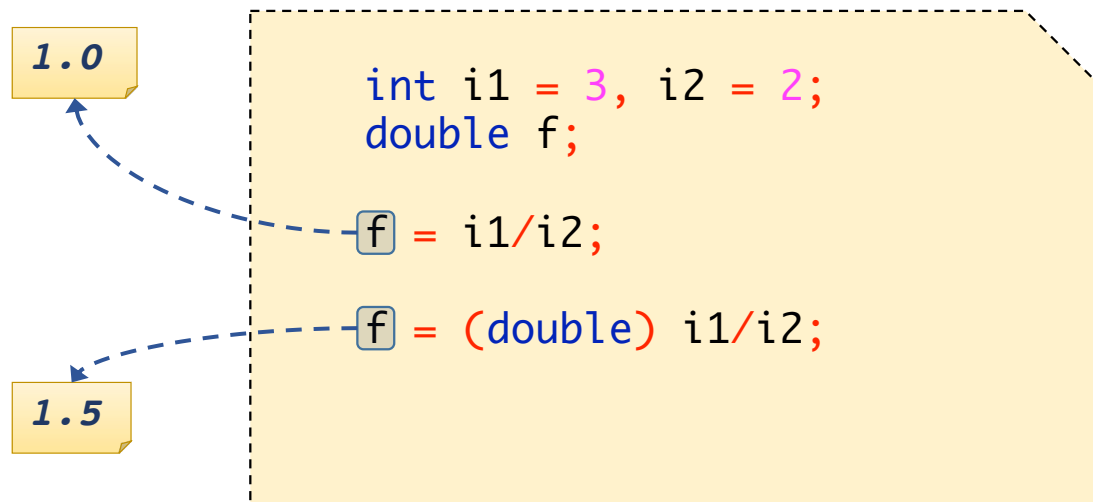
    return 0;
}
```

Conversões Explícitas

- Quando o programador especifica explicitamente o tipo de dado que deve ser utilizado, convertendo-o do seu tipo original.
 - Basta colocar o tipo entre parênteses antes do operando.
 - Esse tipo entre parênteses é o **operador de conversão explícita**.

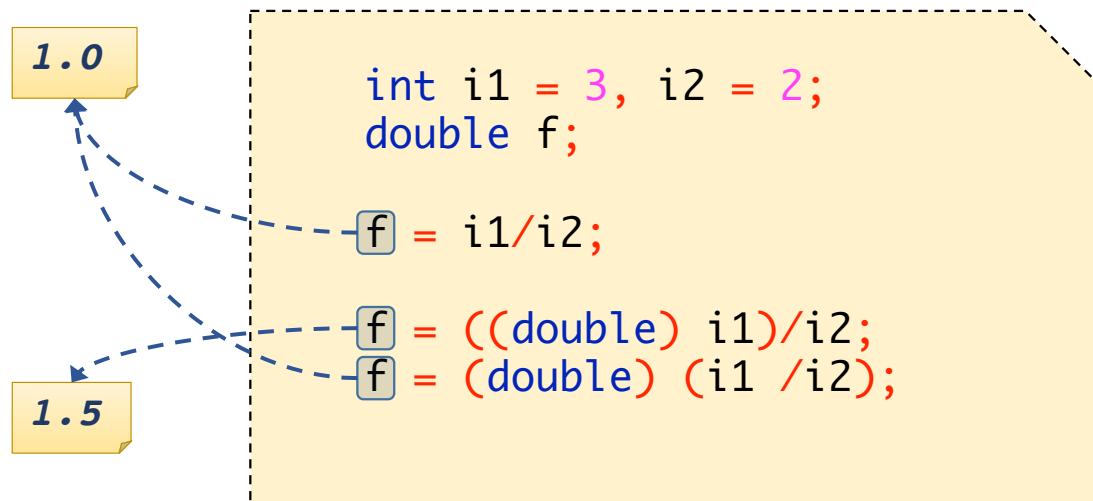
Conversões Explícitas

- Quando o programador especifica explicitamente o tipo de dado que deve ser utilizado, convertendo-o do seu tipo original.
 - Basta colocar o tipo entre parênteses antes do operando.
 - Esse tipo entre parênteses é o **operador de conversão explícita**.



Conversões Explícitas

- Quando o programador especifica explicitamente o tipo de dado que deve ser utilizado, convertendo-o do seu tipo original.
 - Basta colocar o tipo entre parênteses antes do operando.
 - Esse tipo entre parênteses é o **operador de conversão explícita**.



Operadores de Incremento e Decremento

- Prefixado ou Sufixado.
- Efeito Colateral.
- Resultado.
- Em variáveis numéricas ou ponteiros.

OPERAÇÃO	DENOMINAÇÃO	VALOR DA EXPRESSÃO	EFEITO COLATERAL
x++	incremento sufixo	o mesmo de x	adiciona 1 a x
++x	incremento prefixo	o valor de x mais 1	adiciona 1 a x
x--	decremento sufixo	o mesmo de x	subtrai 1 de x
--x	decremento prefixo	o valor de x menos 1	subtrai 1 de x

Operadores de Incremento e Decremento

- Prefixado ou Sufixado.

Antes ou depois da variável.

- Efeito Colateral.
- Resultado.
- Em variáveis numéricas ou ponteiros.

OPERAÇÃO	DENOMINAÇÃO	VALOR DA EXPRESSÃO	EFEITO COLATERAL
x++	incremento sufixo	o mesmo de x	adiciona 1 a x
++x	incremento prefixo	o valor de x mais 1	adiciona 1 a x
x--	decremento sufixo	o mesmo de x	subtrai 1 de x
--x	decremento prefixo	o valor de x menos 1	subtrai 1 de x

Operadores de Incremento e Decremento

- Prefixado ou Sufixado.
- Efeito Colateral.
- Resultado.
- Em variáveis numéricas ou ponteiros.

O operador de incremento `++` sempre vai incrementar a variável em `1` unidade, independente de na forma prefixada ou sufixada. O mesmo vale para o operador de decremento `--`, que sempre vai decrementar em `1` unidade.

OPERAÇÃO	DENOMINAÇÃO	VALOR DA EXPRESSÃO	EFEITO COLATERAL
<code>x++</code>	incremento sufixo	o mesmo de <code>x</code>	adiciona 1 a <code>x</code>
<code>++x</code>	incremento prefixo	o valor de <code>x</code> mais 1	adiciona 1 a <code>x</code>
<code>x--</code>	decremento sufixo	o mesmo de <code>x</code>	subtrai 1 de <code>x</code>
<code>--x</code>	decremento prefixo	o valor de <code>x</code> menos 1	subtrai 1 de <code>x</code>

Operadores de Incremento e Decremento

- Prefixado ou Sufixado.
- Efeito Colateral.
- Resultado.
- Em variáveis numéricas ou ponteiros.

O resultado depende se está sendo utilizado o operador na forma prefixada ou sufixada. Na **prefixada** o resultado é o valor da variável **após** a aplicação do efeito colateral e na **sufixada** é o valor da variável **antes** do efeito colateral.

OPERAÇÃO	DENOMINAÇÃO	VALOR DA EXPRESSÃO	EFEITO COLATERAL
x++	incremento sufixo	o mesmo de x	adiciona 1 a x
++x	incremento prefixo	o valor de x mais 1	adiciona 1 a x
x--	decremento sufixo	o mesmo de x	subtrai 1 de x
--x	decremento prefixo	o valor de x menos 1	subtrai 1 de x

Operadores de Incremento e Decremento

- Que valores são atribuídos à **x** e **y**?

```
int y, x = 2;
```

```
y = 5*x++;
```

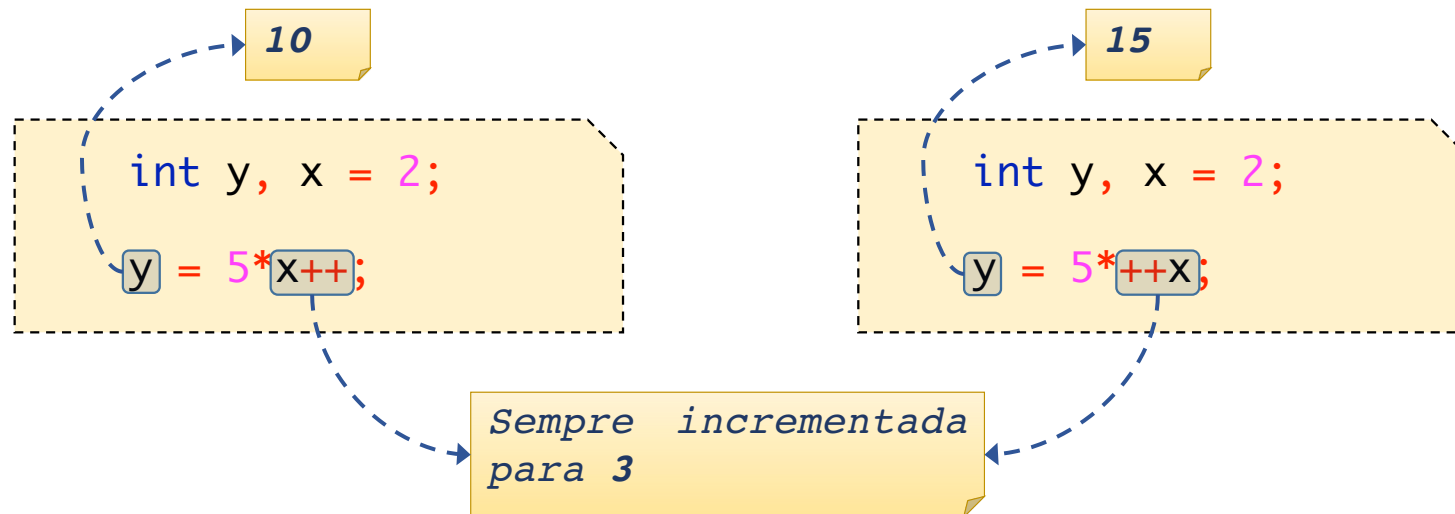
```
int y, x = 2;
```

```
y = 5*++x;
```

[Vídeo](#) com exemplos.

Operadores de Incremento e Decremento

- Que valores são atribuídos à **x** e **y**?



Operadores de Incremento e Decremento

- O valor de ***d*** vai ser sempre incrementado para **2**?

```
int a, b, c, d = 1;
if ((a > b) && (c == d++)){
    a = b = c = 0;
}
```

Operadores de Incremento e Decremento

- O valor de ***d*** vai ser sempre incrementado para **2**?

```
int a, b, c, d = 1;  
if ((a > b) && (c == d++)){  
    a = b = c = 0;  
}
```

Não! Pode ocorrer *curto circuito* do operador **&&**.

Operadores de Incremento e Decremento

- Em qual trecho o valor de ***d*** vai ser sempre incrementado para **2**?

```
int a = 1, b = 2, c = 3, d = 1;  
  
if ((a > b) && (c == d++)){  
    a = b = c = 0;  
}
```

```
int a = 5, b = 2, c = 3, d = 1;  
  
if ((a > b) && (c == d++)){  
    a = b = c = 0;  
}
```

Operadores de Incremento e Decremento

- Em qual trecho o valor de ***d*** vai ser sempre incrementado para **2**?

```
int a = 1, b = 2, c = 3, d = 1;  
if ((a > b) && (c == d++)){  
    a = b = c = 0;  
}
```

```
int a = 5, b = 2, c = 3, d = 1;  
if ((a > b) && (c == d++)){  
    a = b = c = 0;  
}
```

Nesse.

Fluxo de Execução de um Programa

Fluxo de Execução de um Programa

- O **fluxo de execução** de um programa diz respeito a ordem e o número de vezes que cada instrução é executada.
- Um programa com **fluxo natural de execução** é executado sequencialmente da primeira à última instrução, sendo cada uma delas executada uma única vez.
- As **estruturas de controle** podem alterar o fluxo natural de execução de um programa.
 - Os **laços de repetição** alteram a **frequência** de execução das instruções.
 - Os **desvios** alteram a **sequencia** de execução das instruções.

Sequências de Instruções

- Uma instrução em C pode ser:
 - expressão;
 - instrução ***return***;
 - estrutura de controle; ou
 - chamada de função

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

```
int main(void) {  
    int x = 0, y = 0;
```

```
    /* A seguinte instrução é legal, mas não faz sentido */  
    2 * (x + 1);
```

```
    /* A seguinte instrução é legal e faz sentido */  
    y = 2 * (y + 1);
```

```
    printf("\nx = %d, y = %d\n", x, y);
```

```
    return 0;
```

```
}
```



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

```
int main(void) {  
    int x = 0, y = 0;
```

```
    /* A seguinte instrução é legal, mas não faz sentido */  
    2 * (x + 1);
```

```
    /* A seguinte instrução é legal e faz sentido */
```

```
    y = 2 * (y + 1);
```

```
    printf("\nx = %d, y = %d\n", x, y);
```

```
    return 0;
```

```
}
```

Expressões só fazem sentido ou têm utilidade se contiverem operadores com efeito colateral.

Sequências de Instruções

- Uma instrução em C pode ser:
 - expressão;
 - instrução ***return***;
 - estrutura de controle; ou
 - chamada de função.
- **Instruções** podem aparecer **apenas dentro de uma função**.
 - Como a função ***main***.

```
#include <stdio.h>

#define CONSTANCE 42

int x = 0;

x = 2 * x;

int main(void) {
    printf("x = %d", x);

    return 0;
}
```

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

```
#define CONSTANCE 42
```

```
int x = 0;
```

```
x = 2 * x;
```

```
int main(void) {  
    printf("x = %d", x);  
  
    return 0;  
}
```

Definições de variáveis, constantes simbólicas e outros elementos podem aparecer fora do corpo de funções.

Expressões só podem aparecer dentro do corpo de alguma função. Esse código está **errado**.

Sequências de Instruções

- Uma instrução em C pode ser:
 - expressão;
 - instrução ***return***;
 - estrutura de controle; ou
 - chamada de função.
- **Instruções** podem aparecer **apenas dentro de uma função**.
 - Como a função ***main***.
- Toda instrução deve terminar com o **terminador de instruções**
 - O ponto e vírgula ;

Sequências de Instruções

- Uma sequência de instruções, ou **bloco de instruções**, consiste em um conjunto de instruções confinadas entre chaves **{ }**
- Variáveis definidas dentro de blocos só podem ser vistas e utilizadas dentro desse bloco.
 - São chamadas **variáveis locais**.
- Variáveis definidas fora de um bloco, portanto fora de funções, geralmente podem ser vistas e utilizadas em todos os blocos de um programa.
 - São chamadas **variáveis globais**.

Sequências de Instruções

```
{  
    ... /* Primeiro bloco */  
    {  
        ... /* Segundo bloco aninhado dentro do primeiro */  
        {  
            ... /* Terceiro bloco aninhado dentro do segundo */  
        }  
        ...  
    }  
    ...  
}
```

Instruções Vazias

```
#include <stdio.h>

int main(void) {
    int x = 1;

    x = x * 2;
    ;

    x += 3;

    printf("x = %d", x);

    return 0;
}
```


Instruções Vazias

```
#include <stdio.h>

int main(void) {
    int x = 1;

    x = x * 2;
    ;

    x += 3;

    printf("x = %d", x);

    return 0;
}
```

Instruções Vazias

```
#include <stdio.h>

int main(void) {
    int x = 1;

    x = x * 2;
    ;

    x += 3;;

    printf("x = %d", x);;

    return 0;
}
```

Instruções Vazias

```
#include <stdio.h>

int main(void) {
    int x = 1;

    while(x++ < 10);{
        printf("x = %d", x);
    }

    return 0;
}
```

Instruções Vazias

```
#include <stdio.h>

int main(void) {
    int x = 1;

    while(x++ < 10);{
        printf("x = %d", x);
    }

    return 0;
}
```

Atenção: Nesse exemplo o corpo do laço é apenas a instrução vazia!