

# Vetores, *strings* e matrizes

- Vetores

- Série de variáveis do mesmo tipo em endereços contíguos;
- Evitar declaração de várias variáveis.
- *tipo nome [ numero\_de\_elementos ]*

# Vetores, *strings* e matrizes

- Vetores

- Exemplo: *int vetor\_novo[5];*
- O número de elementos deve ser um valor constante.



# Vetores, *strings* e matrizes

- Vetores – inicialização:

- *int vetor\_novo[5] = {3, 4, 78, 678, 20};*
- *int vetor\_novo[ ] = {3, 4, 78, 678, 20};*

	0	1	2	3	4
vetor_novo	3	4	78	678	20
	int	int	int	int	int

# Vetores, *strings* e matrizes

- Vetores – inicialização e acesso a elementos:

```
#include <stdio.h>

void main()
{
    int v[5] = {7, 3, 32, 45, 6};

    printf("%d %d %d %d %d",
        v[0], v[1], v[2], v[3], v[4]);
}
```

7 3 32 45 6

# Vetores, *strings* e matrizes

- Vetores – inicialização e acesso a elementos:

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

```
void main()
```

```
{
```

```
    int v[ ] = {7, 3, 32, 45, 6};
```

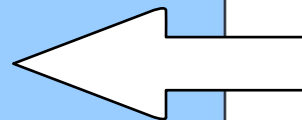
```
    v[0] = 315;
```

```
    v[3] = 723;
```

```
    printf("%d %d %d %d %d",
```

```
    v[0], v[1], v[2], v[3], v[4]);
```

```
}
```



0	1	2	3	4
7	3	32	45	6

# Vetores, *strings* e matrizes

- Vetores – inicialização e acesso a elementos:

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

```
void main()
```

```
{
```

```
    int v[ ] = {7, 3, 32, 45, 6};
```

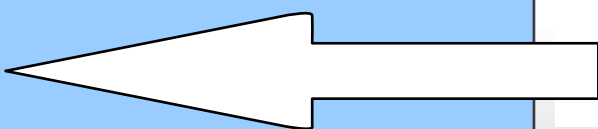
```
    v[0] = 315;
```

```
    v[3] = 723;
```

```
    printf("%d %d %d %d %d",
```

```
    v[0], v[1], v[2], v[3], v[4]);
```

```
}
```



	0	1	2	3	4
v	315	3	32	45	6

# Vetores, *strings* e matrizes

- Vetores – inicialização e acesso a elementos:

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

```
void main()
```

```
{
```

```
    int v[ ] = {7, 3, 32, 45, 6};
```

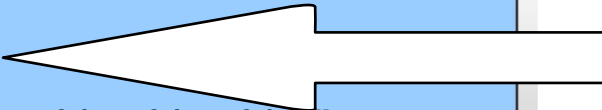
```
    v[0] = 315;
```

```
    v[3] = 723;
```

```
    printf("%d %d %d %d %d",
```

```
    v[0], v[1], v[2], v[3], v[4]);
```

```
}
```



0	1	2	3	4
315	3	32	723	6

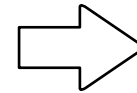
# Vetores, *strings* e matrizes

- Vetores – inicialização e acesso a elementos:

```
#include <stdio.h>

void main()
{
    int v[ ] = {7, 3, 32, 45, 6};

    v[0] = 315;
    v[3] = 723;
    printf("%d %d %d %d %d",
        v[0], v[1], v[2], v[3], v[4]);
}
```



315 3 32 723 6



# Vetores, *strings* e matrizes

## ● Vetores

- Nada impede o programador de acessar uma posição além daquelas definidas na declaração do vetor.

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

```
void main()
```

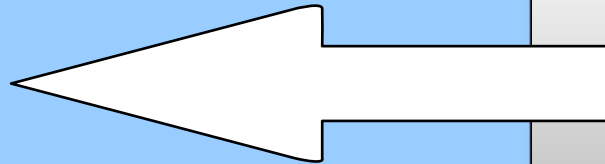
```
{
```

```
int v[ 5 ] = {7, 3, 32, 45, 6};
```

```
v[ 5 ] = 315;
```

```
v[ 10 ] = 723;
```

```
}
```



Não permita que  
isso aconteça!  
Não se pode prever  
os erros decorrentes!!!

# Vetores, *strings* e matrizes

- *Strings*

- Tabela ASCII: caracteres de texto são representados por valores hexadecimais (8 bits)

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

# Vetores, *strings* e matrizes

## ● *Strings*

- Para armazenar uma letra em ASCII, é necessário um byte: uma variável do tipo char.
- Para não armazenarmos várias letras de um texto em muitas variáveis diferentes, utilizamos um vetor de chars.
- Exemplo: `char palavra [ 5 ] = { 'T', 'e', 'x', 't', 'o' };`

	0	1	2	3	4
palavra	'T'	'e'	'x'	't'	'o'
	char	char	char	char	char

# Vetores, *strings* e matrizes

- *Strings*

- Podemos armazenar palavras menores neste vetor. Para indicar o final da palavra, utiliza-se o caracter '\0'.

	0	1	2	3	4
palavra	'O'	'l'	'a'	'\0'	'o'

# Vetores, *strings* e matrizes

## ● *Strings*

- Para simplificar, pode-se usar o seguinte método:
- `char palavra[] = { 'A', 'B', 'C', 'D', '\0' };`
- `char palavra[] = "ABCD";`
- O resultado é o mesmo em ambos:

	0	1	2	3	4
palavra	'A'	'B'	'C'	'D'	'\0'

# Vetores, *strings* e matrizes

- *Strings*

```
#include <stdio.h>
void main()
{
    char v[3];

    v = "Oi";
    v[ ] = "Oi";
    v = { 'O', 'i', '\0' };
    v[ ] = { 'O', 'i', '\0' };
}
```

```
#include <stdio.h>
void main()
{
    char v[3];

    v[0] = 'O';
    v[1] = 'i';
    v[2] = '\0';
}
```

# Vetores, *strings* e matrizes

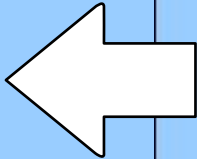
## • *Strings*

- Ambos os métodos são válidos somente na inicialização do vetor.

```
#include <stdio.h>
void main()
{
    char v[3];

    v = "Oi";
    v[ ] = "Oi";
    v = { 'O', 'i', '\0' };
    v[ ] = { 'O', 'i', '\0' };
}
```

*Todos  
errados!!!*



```
#include <stdio.h>
void main()
{
    char v[3];

    v[0] = 'O';
    v[1] = 'i';
    v[2] = '\0';
}
```

*Correto, pois  
trabalha com  
cada elemento  
do vetor.*

# Vetores, *strings* e matrizes

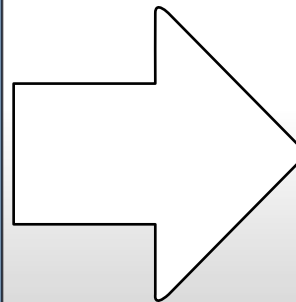
- *Strings* – Visualização

- A função printf() deve ser informada de que vai apresentar um caracter em ASCII através do símbolo %c:

```
#include <stdio.h>

void main()
{
    char v[ ] = "Oi";

    printf("%d %d %d",
           v[0], v[1], v[2]);
}
```



79 105 0



# Vetores, *strings* e matrizes

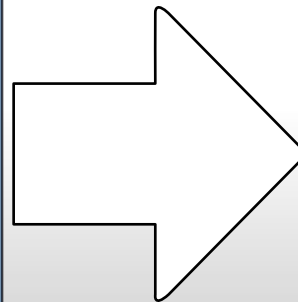
- *Strings* – Visualização

- A função printf() deve ser informada de que vai apresentar um caracter em ASCII através do símbolo %c:

```
#include <stdio.h>

void main()
{
    char v[ ] = "Oi";

    printf("%c %c %c",
           v[0], v[1], v[2]);
}
```



*O i*

# Vetores, *strings* e matrizes

## ● *Strings* – Visualização

- Uma maneira mais simples do que acessar cada elemento do vetor é utilizar o símbolo %s:

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

```
void main()
```

```
{
```

```
    char v[ ] = "Oi";
```

```
    printf("%d %d %d\n",
```

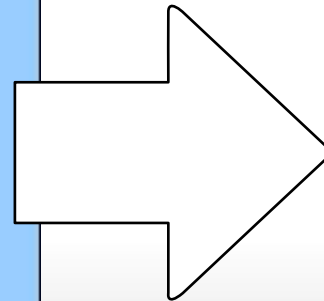
```
           v[0], v[1], v[2]);
```

```
    printf("%c %c %c\n",
```

```
           v[0], v[1], v[2]);
```

```
    printf("%s", v);
```

```
}
```



79 105 0

O i

Oi

A palavra a ser visualizada  
deverá terminar em '\0'

# Vetores, *strings* e matrizes

- Matrizes

- Vetores de vetores - vetores multidimensionais;
- Útil para armazenar tabelas, *pixels* de imagens e videos etc.
- *tipo nome [ num\_elem\_dim1 ] ... [ num\_elem\_dimN ]*

# Vetores, *strings* e matrizes

- Matrizes

- Exemplo: `int vetor_novo[3][4];`

	0	1	2	3
vetor_novo 0				
1				
2				

- O número de elementos também deve ser um valor constante.

# Vetores, *strings* e matrizes

- Matrizes - acesso a elementos

- `vetor_novo[1][2];`

	0	1	2	3
vetor_novo 0				
1				
2				

# Vetores, *strings* e matrizes

- Matrizes

- É possível ter mais de 2 dimensões:
- Exemplo: `char seculo[100][365][24][60][60];`
- Atenção!!! A variável do exemplo acima ocupa  
 $100 \times 365 \times 24 \times 60 \times 60$  bytes = **3,1536 Gigabytes de memória**

# Vetores, *strings* e matrizes

- Matrizes

- Matrizes são simplesmente uma abstração para programadores.

- Por exemplo, utilizar variáveis

*char mat1[3][4];*

ou

*char mat2[12];*

oferece os mesmos resultados.

- Muda somente a forma de acessar as posições na matriz.

# Vetores, *strings* e matrizes

- Matrizes

- Exemplo: *char mat1[3][4];* e *char mat2[12];*

		0	1	2	3
mat1	0	mat2[0]	mat2[1]	mat2[2]	mat2[3]
	1	mat2[4]	mat2[5]	mat2[6]	mat2[7]
	2	mat2[8]	mat2[9]	mat2[10]	mat2[11]



# Operadores

- Atribuição

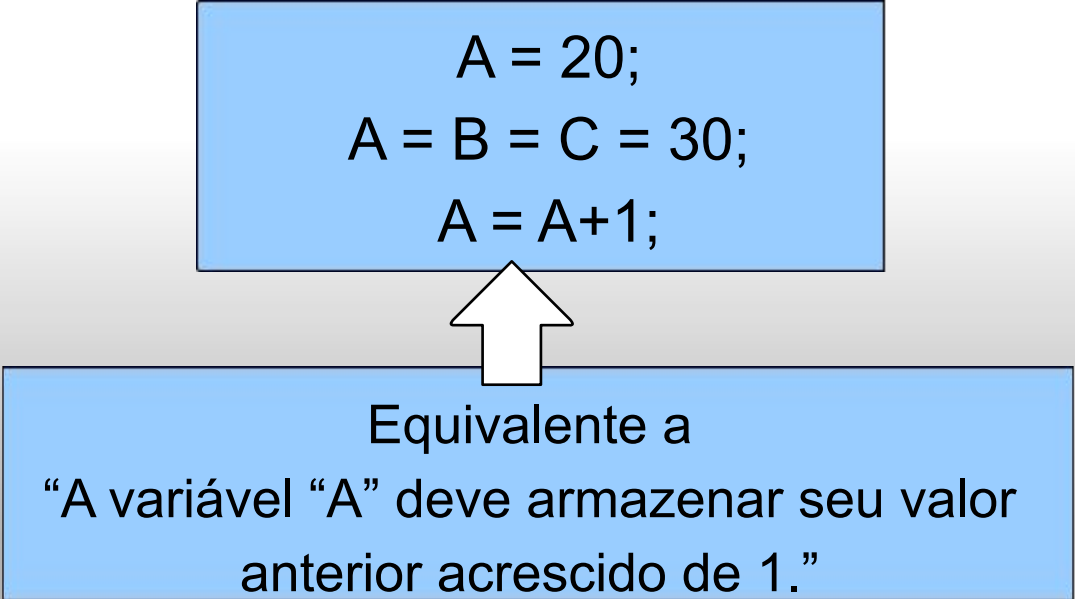
- Sinal de igual (=)
- Como visto anteriormente, definimos valores de variáveis utilizando este operador.

```
A = 20;  
A = B = C = 30;  
A = A+1;
```

# Operadores

- Atribuição

- Sinal de igual (=)
- Como visto anteriormente, definimos valores de variáveis utilizando este operador.



```
A = 20;  
A = B = C = 30;  
A = A+1;
```

The diagram consists of two light blue rectangular boxes with black borders. The top box contains three lines of code: 'A = 20;', 'A = B = C = 30;', and 'A = A+1;'. A white arrow with a black outline points upwards from the bottom box to the bottom center of the top box. The bottom box contains the text 'Equivalente a' followed by a quote: '“A variável “A” deve armazenar seu valor anterior acrescido de 1.”'.

Equivalente a

“A variável “A” deve armazenar seu valor anterior acrescido de 1.”

# Operadores

- Aritmética

+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão

```
A = 3;  
B = A+15;  
C = B*A;  
D = C/9;  
E = D%5;
```



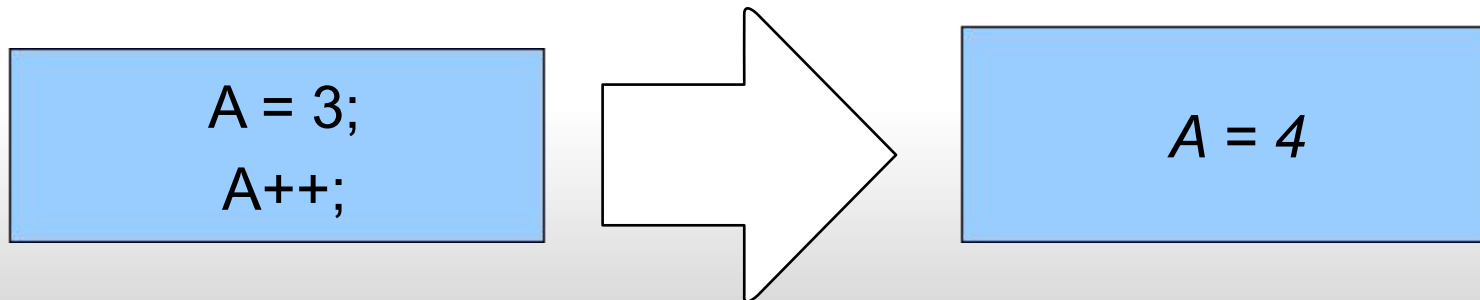
```
A = 3  
B = 18  
C = 54  
D = 6  
E = 1
```

# Operadores

- Aritmética

++	Adição simplificada
--	Subtração simplificada

Servem como prefixos  
ou sufixos

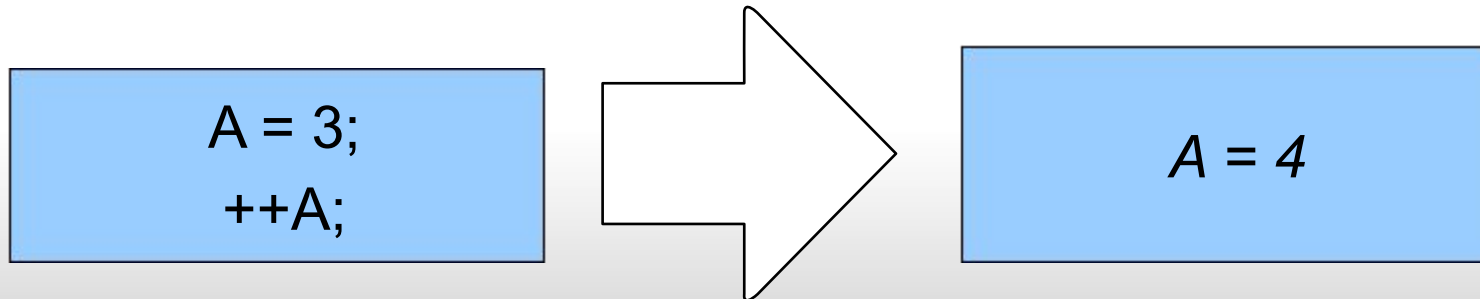


# Operadores

- Aritmética

++	Adição simplificada
--	Subtração simplificada

Servem como prefixos  
ou sufixos

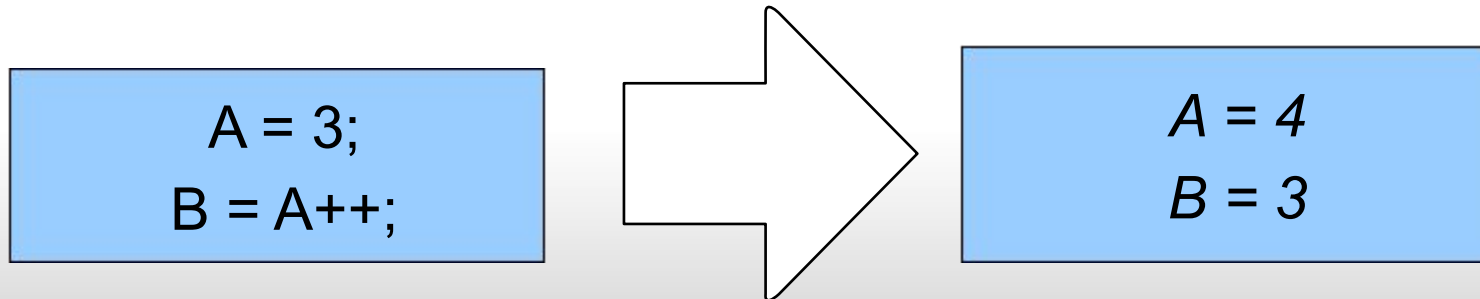


# Operadores

- Aritmética

++	Adição simplificada
--	Subtração simplificada

Servem como prefixos  
ou sufixos



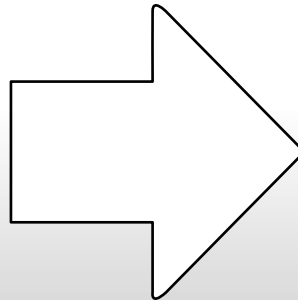
# Operadores

- Aritmética

++	Adição simplificada
--	Subtração simplificada

Servem como prefixos  
ou sufixos

A = 3;  
B = ++A;



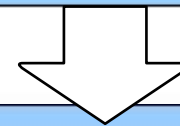
A = 4  
B = 4

# Operadores

- Relações e igualdades

==	Igual a
!=	Diferente de
>	Maior que
<	Menor que
>=	Maior ou igual que
<=	Menor ou igual que

```
A = 3;  
B = 15;  
C = (A==B);  
D = (A!=B);  
E = (A>B);  
F = (A<B);  
G = (A>=B);  
H = (A<=B);
```



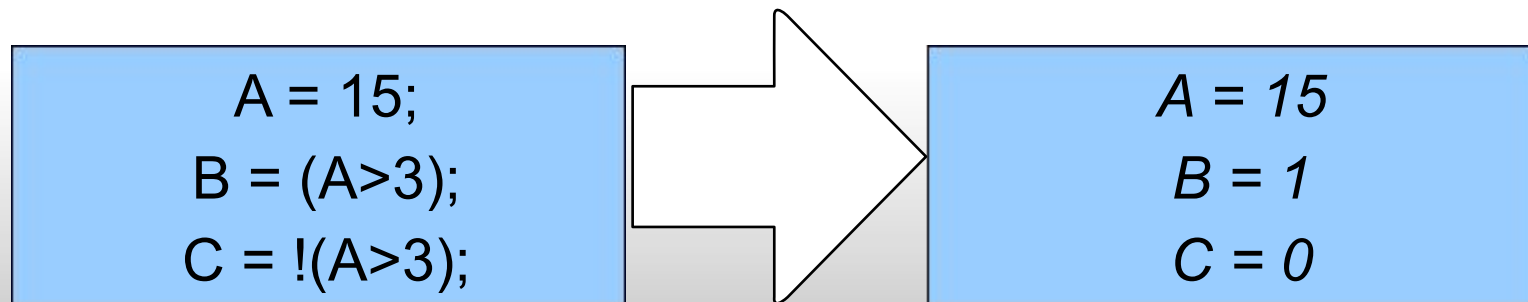
```
C = 0  
D = 1  
E = 0  
etc...
```



# Operadores

- Operadores lógicos – negação (!)

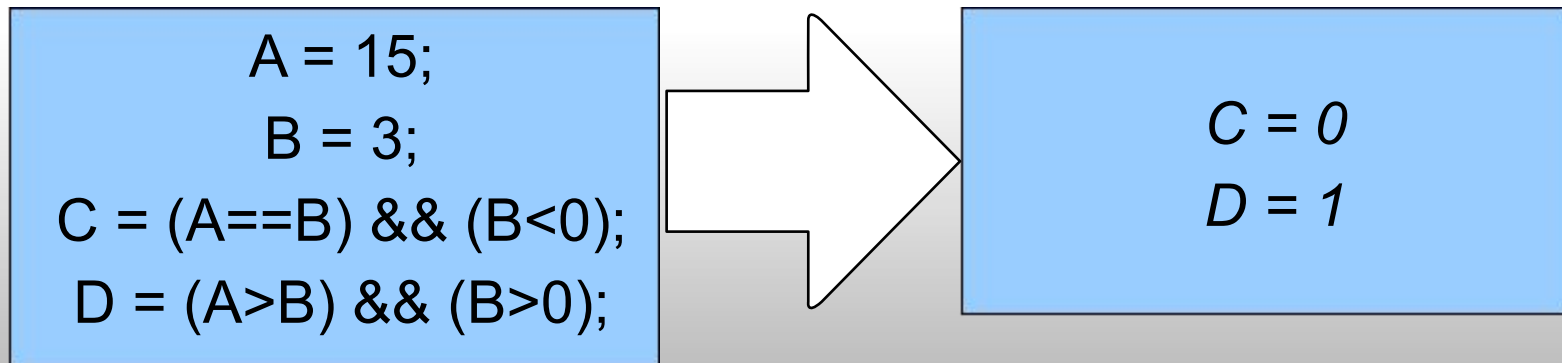
A	!A
0	1
1	0



# Operadores

- Operadores lógicos – E (&&)

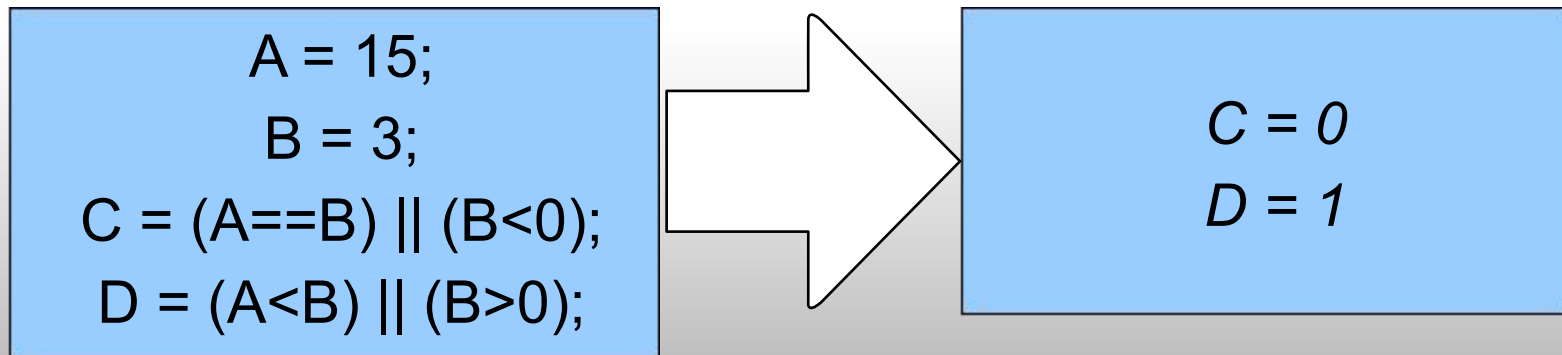
A	B	A&&B
0	0	0
0	1	0
1	0	0
1	1	1



# Operadores

- Operadores lógicos – OU (||)

A	B	A  B
0	0	0
0	1	1
1	0	1
1	1	1



# Operadores

- Operadores lógicos *bit a bit*

&	E
	OU
^	XOR (OU EXCLUSIVO)
~	NEGAÇÃO
>>	Deslocamento de bits à direita
<<	Deslocamento de bits à esquerda

# Operadores

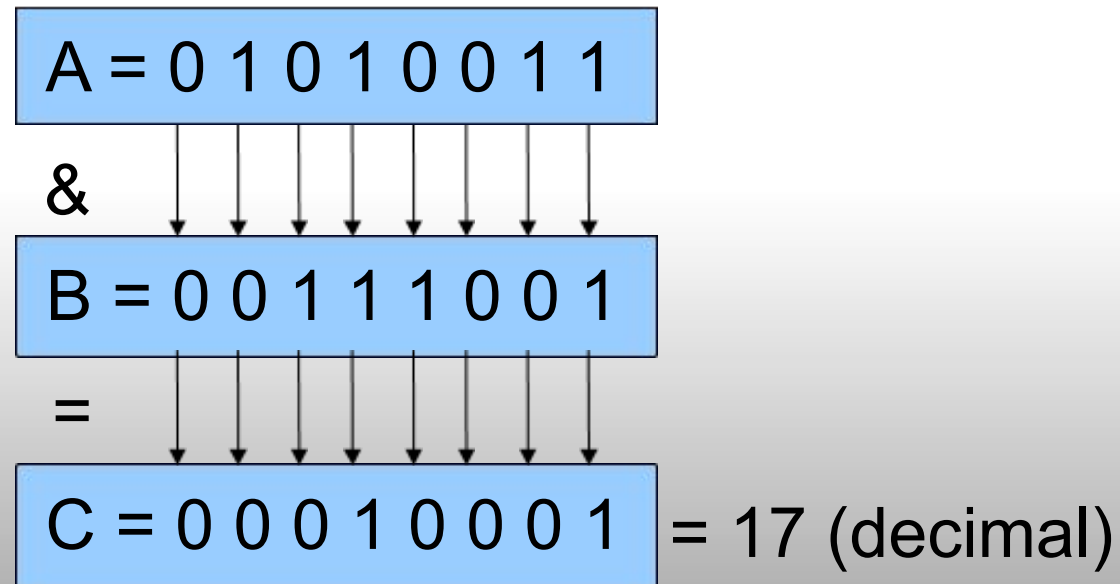
- Operadores lógicos *bit a bit* – exemplo

- char A = 8 (em base decimal)
- A = 00001000 (em base binária)
- A>>1 = 00000100 (binário) = 4 (decimal)
- A<<3 = 00100000 (binário) = 32 (decimal)
- A<<6 = 00000001 (binário) = 1 (decimal)

# Operadores

- Operadores lógicos *bit a bit* – exemplo

- char A = 83 (decimal) = 01010011 (binário)
- char B = 57 (decimal) = 00111001 (binário)
- char C = A & B = (01010011) & (00111001)

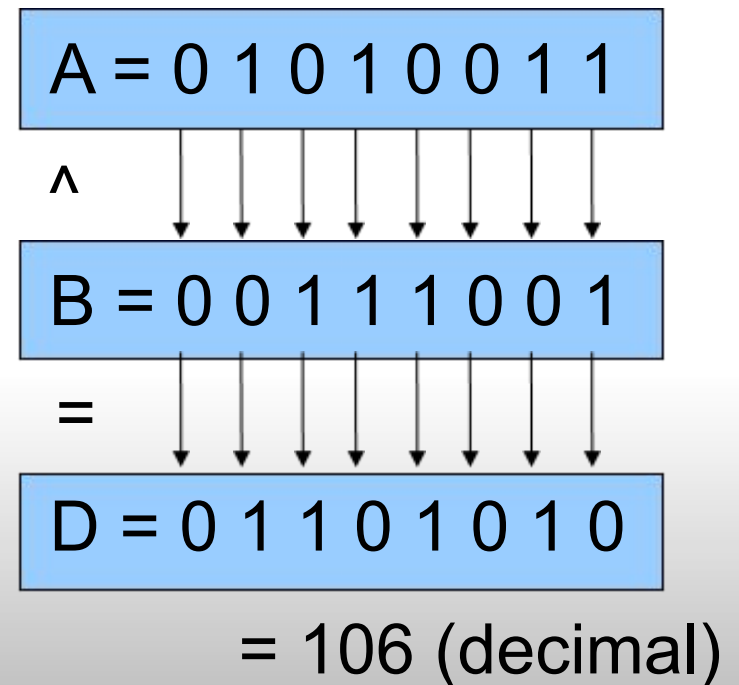


# Operadores

- Operadores lógicos *bit a bit* – exemplo

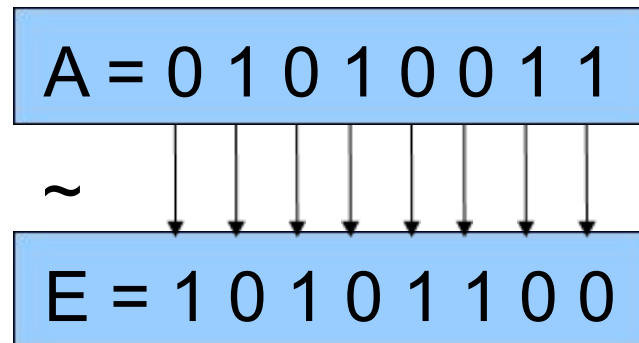
- char D = A ^ B = (01010011) ^ (00111001)

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0



# Operadores

- Operadores lógicos *bit a bit* – exemplo
  - char E =  $\sim A = \sim (01010011)$



= 172 (decimal)



# Operadores

- Expressões abreviadas

- $A = 3;$
- A expressão “ $A = A+10;$ ” pode ser escrita como “ $A+=10;$ ”
- A mesma técnica é válida para os operadores  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $>>$ ,  $<<$ ,  $\&$ ,  $|$  e  $\wedge$
- Por exemplo:  
     $f = 10; f \% = 9;$   
    resulta em  $f = 10\%9 = 1$

# Operadores

- Operador condicional

- *CondiçãoX ? Resultado1 : Resultado*
  - Significa “Se a CondiçãoX for verdadeira, retorne o valor Resultado1. Caso contrário, retorne o valor Resultado2.”
  - Por exemplo:
    - $a = 10;$
    - $b = (a > 20) ? 2 : 5;$
    - $c = (a == 10) ? a + 5 : a + 7$
- resulta em  $a = 10$ ,  $b = 5$  e  $c = 15$

# Operadores

- Modeladores

- A existência de diferentes tipos de variáveis pode causar erros.
- Por exemplo:  
    unsigned char A;  
    A = 256;
- O número 256 em decimal é escrito 100000000 em binário. São 9 *bits*. Como a variável *A* só tem 8 *bits* (por ser *unsigned char*), ela receberá o valor 00000000 (os 8 *bits* menos significativos).

# Operadores

- Modeladores

- Caso a variável seja *signed*, isto poderá gerar ainda mais problemas.
- Problemas semelhantes podem surgir quando misturamos diferentes variáveis.
- Por exemplo, se temos três valores inteiros, e queremos saber a média aritmética dos três, é bem provável que esta terá um valor de ponto flutuante.

# Operadores

- Modeladores

- Para evitar erros na conta devido a conflitos na representação dos valores, utilizamos modeladores.

```
int a = 30, b = 10, c = 6;
```

```
float media;
```

```
media = (float)a;
```

```
media += (float)b;
```

```
media += (float)c;
```

```
media /= 3.0;
```

# Ponteiros

- Variáveis:
  - Espaços na memória para guardar informações;
  - Recebem nomes únicos para identificação;
  - Não ligamos para sua localização física em si.

# Ponteiros

- Variáveis:
  - As linguagens C e C++ apresentam uma série de situações em que a localização física é importante.
  - Ponteiros são variáveis que indicam a localização física de outras variáveis.

# Ponteiros

- Memória do computador:
  - 'Células' de *bytes* sucessivos → Posições únicas.





# Ponteiros

- Ponteiros - declaração
  - *tipo \*nome\_do\_ponteiro;*
  - Exemplo: *char \*pont;* é um ponteiro que pode apontar para variáveis do tipo char.

# Ponteiros

- Ponteiros - operadores & e \*
  - $\&var1$  = endereço de variável *var1*
  - $*var2$  = valor da variável **APONTADA** pela variável *var2*

# Ponteiros

- Ponteiros

```
#include <stdio.h>

void main( )
{
    char a;
    char *b;

    a = 23;
    b = &a;
    printf("Antes, a = %d\n",a);
    *b = 33;
    printf("Depois, a = %d\n",a);
}
```

# Ponteiros

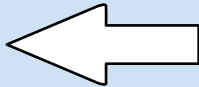
- Ponteiros

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

```
void main( )  
{
```

```
    char a;  
    char *b;
```

```
    a = 23;
```



a recebe o valor 23

```
    b = &a;
```

```
    printf("Antes, a = %d\n",a);
```

```
    *b = 33;
```

```
    printf("Depois, a = %d\n",a);
```

```
}
```

# Ponteiros

- Ponteiros

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

```
void main( )  
{
```

```
    char a;  
    char *b;
```

```
    a = 23;
```

```
    b = &a;
```

```
    printf("Antes, a = %d", a);
```

```
    *b = 33;
```

```
    printf("Depois, a = %d\n", a);
```

```
}
```

Assumindo que *a* está na posição 1567,  
*b* recebe o valor 1567

# Ponteiros

- Ponteiros

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

```
void main( )  
{
```

```
    char a;  
    char *b;
```

```
    a = 23;
```

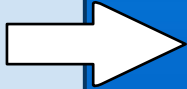
```
    b = &a;
```

```
    printf("Antes, a = %d\n",a);
```

```
    *b = 33;
```

```
    printf("Depois, a = %d\n",a);
```

```
}
```



*Antes, a = 23*

# Ponteiros

- Ponteiros

```
#include <stdio.h>

void main( )
{
    char a;
    char *b;

    a = 23;
    b = &a;
    printf("Antes, a = %d\n", a);
    *b = 33;
    printf("Depois, a = %d\n", a);
}
```

A variável na posição 1567 recebe o valor 33. Isto é, a variável a, por estar na posição 1567, recebe o valor 33.

# Ponteiros

- Ponteiros

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

```
void main( )  
{
```

```
    char a;  
    char *b;
```

Não confundir com o uso do \* na declaração do ponteiro.  
Aqui, o \* está somente indicando que *b* é um ponteiro.

```
    a = 23;
```

```
    b = &a;
```

```
    printf("Antes, a = %d\n", a);
```

```
    *b = 33;
```

```
    printf("Depois, a = %d\n", a);
```

```
}
```

A variável na posição 1567 recebe o valor 33. Isto é, a variável *a*, por estar na posição 1567, recebe o valor 33.



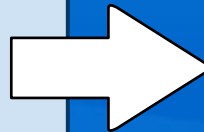
# Ponteiros

- Ponteiros

```
#include <stdio.h>

void main( )
{
    char a;
    char *b;

    a = 23;
    b = &a;
    printf("Antes, a = %d\n",a);
    *b = 33;
    printf("Depois, a = %d\n",a);
}
```



*Antes, a = 23*  
*Depois, a = 33*

# Ponteiros

- Ponteiros

```
#include <stdio.h>

void main( )
{
    char a;
    char *b;

    a = 23;
    b = &a;
    printf("Antes, a = %d\n",a);
    *b = 33;
    printf("Depois, a = %d\n",a);
}
```

Devido a essa capacidade de alterar valores diretamente, é preciso indicar o tipo de dado do ponteiro

*Antes, a = 23*  
*Depois, a = 33*

# Ponteiros

- Aritmética de ponteiros

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

```
void main( )
```

```
{
```

```
    char a;
```

```
    char *b;
```

```
    a = 23;
```

```
    b = &a;
```

```
    b++;
```

```
}
```

# Ponteiros

- Aritmética de ponteiros

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

```
void main( )  
{
```

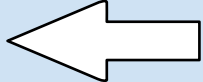
```
    char a;  
    char *b;
```

```
    a = 23;
```

```
    b = &a;
```

```
    b++;
```

```
}
```



Assumindo que *a* está na posição 1567,  
*b* recebe o valor 1567

# Ponteiros

- Aritmética de ponteiros

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

```
void main( )  
{
```

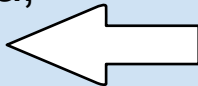
```
    char a;  
    char *b;
```

```
    a = 23;
```

```
    b = &a;
```

```
    b++;
```

```
}
```



O endereço 1567 diz respeito ao *byte* de número 1567. Quando incrementamos *b*, andamos mais um byte na memória (porque *b* é do tipo char, que ocupa um *byte*).

Ou seja, *b* = 1568.

# Ponteiros

- Aritmética de ponteiros

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

```
void main( )
```

```
{
```

```
    int a;
```

```
    int *b;
```

```
    a = 23;
```

```
    b = &a;
```

```
    b++;
```

```
}
```

# Ponteiros

- Aritmética de ponteiros

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

```
void main( )
```

```
{
```

```
    int a;
```

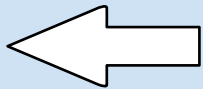
```
    int *b;
```

```
    a = 23;
```

```
    b = &a;
```

```
    b++;
```

```
}
```



Assumindo que *a* está na posição 1567,  
*b* recebe o valor 1567

# Ponteiros

- Aritmética de ponteiros

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

```
void main( )
```

```
{
```

```
    int a;
```

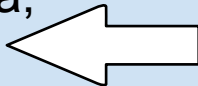
```
    int *b;
```

```
    a = 23;
```

```
    b = &a;
```

```
    b++;
```

```
}
```



Quando incrementamos *b*, andamos mais QUATRO bytes na memória (porque *b* é do tipo int, que ocupa quatro *bytes*).

Ou seja, *b* = 1571.



# Ponteiros

- Aritmética de ponteiros

```
#include <stdio.h>

void main( )
{
    int a[4]; int *b;

    a[0] = 23;
    b = &(a[0]);
    b++;
    *b = 789;
    *(b+1) = 354;
    b = &(a[3]);
    *b = 90;
    printf("a = {%d, %d, %d, %d}",
        a[0], a[1], a[2], a[3]);
}
```

# Ponteiros

- Aritmética de ponteiros

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

```
void main( )
```

```
{
```

```
    int a[4]; int *b;
```

```
    a[0] = 23;
```

```
    b = &(a[0]);
```

```
    b++;
```

```
    *b = 789;
```

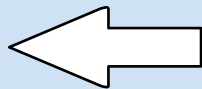
```
    *(b+1) = 354;
```

```
    b = &(a[3]);
```

```
    *b = 90;
```

```
    printf("a = {%d, %d, %d, %d}",  
          a[0], a[1], a[2], a[3]);
```

```
}
```



*b* recebe o endereço de *a[0]*  
*b* = endereço do primeiro elemento do vetor *a*

# Ponteiros

- Aritmética de ponteiros

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

```
void main( )
```

```
{
```

```
    int a[4]; int *b;
```

```
    a[0] = 23;
```

```
    b = &(a[0]);
```

```
    b++;
```

```
    *b = 789;
```

```
    *(b+1) = 354;
```

```
    b = &(a[3]);
```

```
    *b = 90;
```

```
    printf("a = {%d, %d, %d, %d}",  
          a[0], a[1], a[2], a[3]);
```

```
}
```

*b* = endereço de *a*[0] + 4 bytes (tamanho de um int)  
==> *b* = endereço de *a*[1]

# Ponteiros

- Aritmética de ponteiros

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

```
void main( )
```

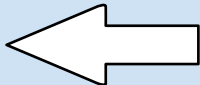
```
{
```

```
    int a[4]; int *b;
```

```
    a[0] = 23;
```

```
    b = &(a[0]);
```

```
    b++;
```

```
    *b = 789; 
```

```
    *(b+1) = 354;
```

```
    b = &(a[3]);
```

```
    *b = 90;
```

```
    printf("a = {%d, %d, %d, %d}",  
          a[0], a[1], a[2], a[3]);
```

```
}
```

*a[1] = 789*

# Ponteiros

- Aritmética de ponteiros

```
#include <stdio.h>

void main( )
{
    int a[4]; int *b;

    a[0] = 23;
    b = &(a[0]);
    b++;
    *b = 789;
    *(b+1) = 354;
    b = &(a[3]);
    *b = 90;
    printf("a = {%d, %d, %d, %d},\n",
           a[0], a[1], a[2], a[3]);
}
```



Guarde o valor 354 no endereço  
apontado por  $b + 4$  bytes  
==> Guarde o valor 354 no endereço de  $a[1] + 4$  bytes  
==>  $a[2] = 354$

# Ponteiros

- Aritmética de ponteiros

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

```
void main( )
```

```
{
```

```
    int a[4]; int *b;
```

```
    a[0] = 23;
```

```
    b = &(a[0]);
```

```
    b++;
```

```
    *b = 789;
```

```
    *(b+1) = 354;
```

```
    b = &(a[3]);
```

```
    *b = 90;
```

```
    printf("a = {%d, %d, %d, %d}",  
          a[0], a[1], a[2], a[3]);
```

```
}
```

*b* = endereço de *a*[3]

==> *b* = endereço da quarta posição do vetor *a*

# Ponteiros

- Aritmética de ponteiros

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

```
void main( )
```

```
{
```

```
    int a[4]; int *b;
```

```
    a[0] = 23;
```

```
    b = &(a[0]);
```

```
    b++;
```

```
    *b = 789;
```


```
    *(b+1) = 354;
```

```
    b = &(a[3]);
```

```
    *b = 90;
```

```
    printf("a = {%d, %d, %d, %d},\n",  
          a[0], a[1], a[2], a[3]);
```

```
}
```



*a[3] = 90*

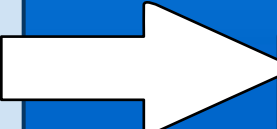
# Ponteiros

- Aritmética de ponteiros

```
#include <stdio.h>

void main( )
{
    int a[4]; int *b;

    a[0] = 23;
    b = &(a[0]);
    b++;
    *b = 789;
    *(b+1) = 354;
    b = &(a[3]);
    *b = 90;
    printf("a = {%d, %d, %d, %d}",
        a[0], a[1], a[2], a[3]);
}
```



*a = {23, 789, 354, 90}*



# Ponteiros

- Ponteiros e vetores
  - Quando declaramos, por exemplo, *char str[20];*, estamos indicando que queremos alocar 20 posições de memória do tamanho char. Quando acessamos *str[13]*, estamos acessando a 14<sup>a</sup> posição do vetor, ou 13 posições além do início do vetor.

# Ponteiros

- Ponteiros e vetores
  - Se criarmos um ponteiro *char \*p\_str;*, e guardarmos nele o endereço de *str[0]*, então as expressões

*str[0] = 0;*

*\*p\_str = 0;*

são equivalentes, bem como

*str[19] = 76;*

*\*(p\_str+19) = 76;*

# Ponteiros

- Ponteiros e vetores
  - Na verdade, o identificador *str* guarda o primeiro endereço da memória de 20 chars que foi alocada. Tanto faz usar

*p\_str = &(str[0]);*

ou

*p\_str = str;*

# Ponteiros

- Ponteiros e vetores
  - Fazer o contrário ( $str = p\_str$ ) não faz sentido.  $str$  já é um espaço de 20 *bytes* alocado na memória, não havendo necessidade de fazê-lo apontar para um outro endereço qualquer ( $p\_str$ ).