

# Programação e Desenvolvimento de Software I

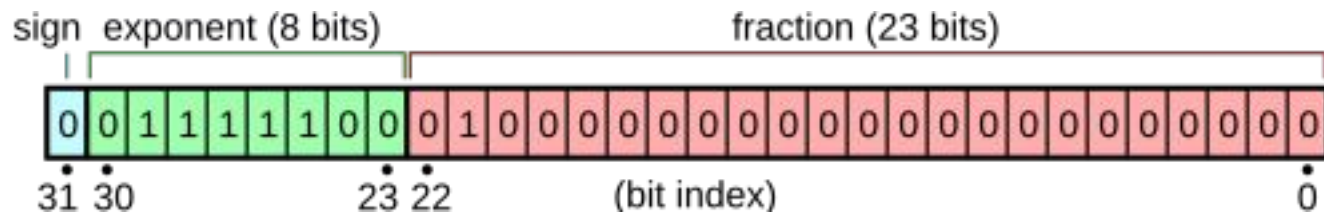
## Ponteiros e operações bit-a-bit

---

Prof. Héctor Azpúrua  
(slides adaptados do Prof. Pedro Olmo)

# Resumo da aula anterior

## Duvidas



- Sobre o tamanho das variáveis e os ranges:

Nome do tipo	Bytes	Range de valores
float	4 (32 bits)	3.4E-38 até 3.4E+38

- Para representar o valor máximo então temos:

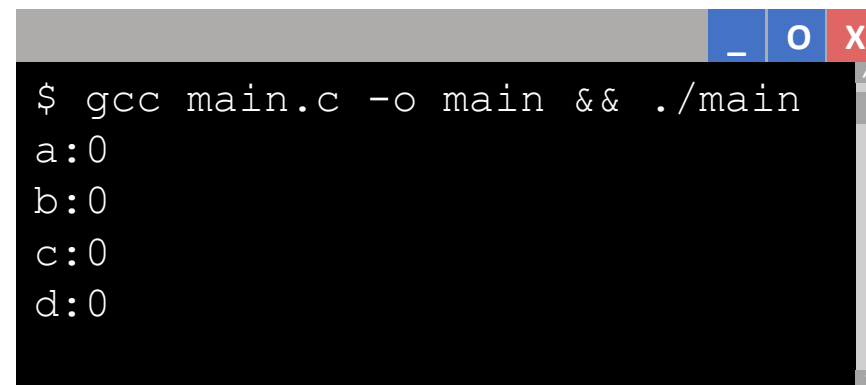
- 1 bit para signo
- 8 bits expoente:  $11111111_2$ 
  - $11111111_2$  é usado para o numero especial NaN
  - $11111110_2 = +127_{10}$
- 23 bits + 1 bit implícito na mantisa (por que sempre começa com 1)
  - $1.111111111111111111111111_2$  (adicionando o 1 implícito no começo)
  - $1.99999988079071044921875_{10}$
- Totalizando:
  - $-(1)^0 \times 2^{+127} \times 1.99999988079071044921875 = 34028234663852885981170418348451692540$   
**3.4E38**

# Resumo da aula anterior

## Duvidas

- A ordem dos modificadores de tipo de variável importam?
  - Não faz diferença

```
int main() {  
    int long a = 0;  
    long int b = 0;  
    unsigned int c = 0;  
    short unsigned int d = 0;  
    printf("a:%ld\n", a);  
    printf("b:%ld\n", b);  
    printf("c:%d\n", c);  
    printf("d:%d\n", d);  
}
```



```
$ gcc main.c -o main && ./main  
a:0  
b:0  
c:0  
d:0
```

# Resumo da aula anterior

## Duvidas

- `long int` pode ser impresso com o tag `%d` do `printf`?
  - Funciona mas não é recomendado, para números maiores tem um tag específico: `%ld`
  - `unsigned` tem um tag específico

// signed

```
int int_var = pow(2, 31) - 1;
```

```
long long int long_long_int_var = pow(2, 63) - 1;
```

```
short int short_int_var = pow(2, 15) - 1;
```

```
printf("Todas as variaveis com signo (%d):\n");
```

```
printf("\tint:%d\n", int_var);
```

```
printf("\tlong_long_int_var:%d\n", long_long_int_var);
```

```
printf("\tshort_int_var:%d\n", short_int_var);
```

```
printf("Fazendo da forma correta:\n");
```

```
printf("\tint (%d):%d\n", int_var);
```

```
printf("\tlong_long_int_var (%ld):%ld\n", long_long_int_var);
```

```
printf("\tshort_int_var:(%hd): %hd\n", short_int_var);
```

```
Todas as variaveis com signo (%d):
```

```
int:2147483647
```

```
long_long_int_var:-1
```

```
short_int_var:32767
```

```
Fazendo da forma correta:
```

```
int (%d):2147483647
```

```
long_long_int_var
```

```
(%ld):9223372036854775807
```

# Resumo da aula anterior

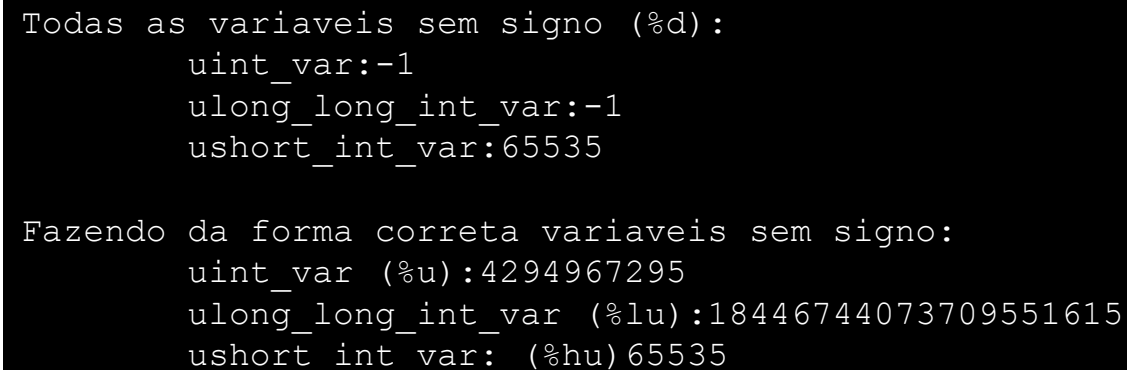
## Duvidas

- `long int` pode ser impresso com o tag `%d` do `printf`?
  - Funciona mas não é recomendado, para números maiores tem um tag específico: `%ld`
  - `unsigned` tem um tag específico

```
// unsigned
unsigned int uint_var = pow(2, 32) - 1;
unsigned long long int ulong_long_int_var = pow(2, 64) - 1;
unsigned short int ushort_int_var = pow(2, 16) - 1;

printf("Todas as variaveis sem signo (%d):\n");
printf("\tuint_var:%d\n", uint_var);
printf("\tulong_long_int_var:%d\n", ulong_long_int_var);
printf("\tushort_int_var:%d\n", ushort_int_var);

printf("Fazendo da forma correta:\n");
printf("\tuint_var (%u):%u\n", uint_var);
printf("\tulong_long_int_var (%lu):%lu\n", ulong_long_int_var);
printf("\tushort_int_var: (%hu)%hu\n", ushort_int_var);
```



```
Todas as variaveis sem signo (%d):
    uint_var:-1
    ulong_long_int_var:-1
    ushort_int_var:65535

Fazendo da forma correta variaveis sem signo:
    uint_var (%u):4294967295
    ulong_long_int_var (%lu):18446744073709551615
    ushort_int_var: (%hu)65535
```

# Modificadores de tipo

[https://en.wikipedia.org/wiki/C\\_data\\_types](https://en.wikipedia.org/wiki/C_data_types)

Nome do tipo	Bytes	Range de valores (decimal)	Tag printf
char	1	-128 até 127	%c (imprime letra)
unsigned char	1	0 até 255	%c (imprime letra)
int	4	-2,147,483,648 até 2,147,483,647	%i ou %d
short int	2	-32,768 até 32,767	%hd
unsigned short int	2	0 até 65535	%hu
unsigned int	4	0 até 4,294,967,295	%u
float	4	3.4E-38 até 3.4E+38	%f
long long int	8	-9,223,372,036,854,775,808 até 9,223,372,036,854,775,807	%lld
double	8	1.7E-308 até 1.7E+308	%lf
long double	10	3.4E-4932 até 1.1E+4932	%Lf

# Modificadores de tipo

[https://en.wikipedia.org/wiki/C\\_data\\_types](https://en.wikipedia.org/wiki/C_data_types)

## Primary types [\[edit\]](#)

### Main types [\[edit\]](#)

The C language provides the four basic arithmetic type specifiers `char`, `int`, `float` and `double` (as well as the boolean type `bool`), and the modifiers `signed`, `unsigned`, `short`, and `long`. The following table lists the permissible combinations in specifying a large set of storage size-specific declarations.

Type	Explanation	Size (bits)	Format specifier	Range	Suffix for decimal constants
<code>bool</code>	Boolean type, added in <a href="#">C23</a> .	1 (exact)	<code>%d</code>	[false, true]	—
<code>char</code>	Smallest addressable unit of the machine that can contain basic character set. It is an <a href="#">integer</a> type. Actual type can be either signed or unsigned. It contains <code>CHAR_BIT</code> bits. <sup>[3]</sup>	≥8	<code>%C</code>	[ <code>CHAR_MIN</code> , <code>CHAR_MAX</code> ]	—
<code>signed char</code>	Of the same size as <code>char</code> , but guaranteed to be signed. Capable of containing at least the [−127, +127] range. <sup>[3][a]</sup>	≥8	<code>%C</code> <sup>[b]</sup>	[ <code>SCHAR_MIN</code> , <code>SCHAR_MAX</code> ] <sup>[6]</sup>	—
<code>unsigned char</code>	Of the same size as <code>char</code> , but guaranteed to be unsigned. Contains at least the [0, 255] range. <sup>[7]</sup>	≥8	<code>%C</code> <sup>[c]</sup>	[0, <code>UCHAR_MAX</code> ]	—
<code>short</code> <code>short int</code> <code>signed short</code>	Short signed integer type. Capable of containing at least the [−32 767, +32 767] range. <sup>[3][a]</sup>	≥16	<code>%hi</code> or <code>%hd</code>	[ <code>SHRT_MIN</code> , <code>SHRT_MAX</code> ]	—
	Contains at least the	≥16	<code>%hu</code>	[0, <code>USHRT_MAX</code> ]	—

- [Signals](#)
- [Alternative tokens](#)

#### Miscellaneous headers

- `<assert.h>`
- `<errno.h>`
- `<setjmp.h>`
- `<stdarg.h>`

V · T · E

# Endereços de variáveis

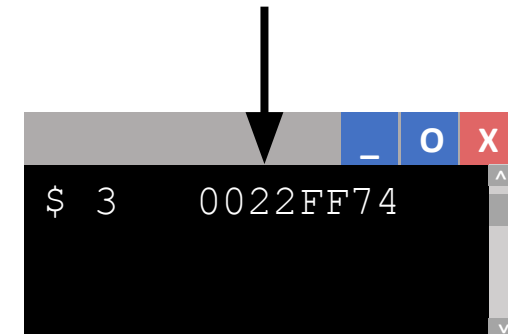
- Uma variável **representa um nome simbólico** para uma **posição de memória**
- Cada posição de memória de um computador possui um endereço
  - Logo, o endereço de uma variável é o endereço da posição de memória representada pela variável

```
int x = 3;  
printf("%d %p", x, &x);
```

Operador para obtenção  
do endereço da variável



Endereço no sistema  
hexadecimal





# Endereços de variáveis

Endereço	Variável	Conteúdo
0022FF70	salario	891
0022FF71	c	'a'
0022FF72	idade	8
0022FF73	velocidade	16.1
0022FF74	x	3
0022FF75	km	298347

# Endereços de variáveis

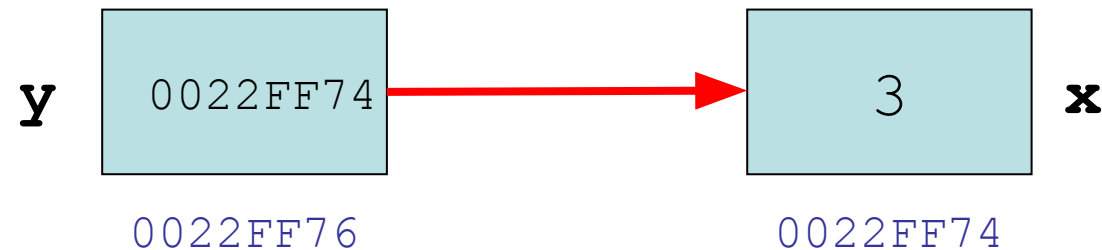
- Note que o endereço de uma variável é um valor
  - Logo, uma variável pode armazenar um endereço
- Uma variável que armazena um endereço de memória é conhecida como **ponteiro (pointer)**
- Daí o porquê do **tag** usado para exibir endereços de memória ser  $\%p$

# Endereços de variáveis

- Exemplo:

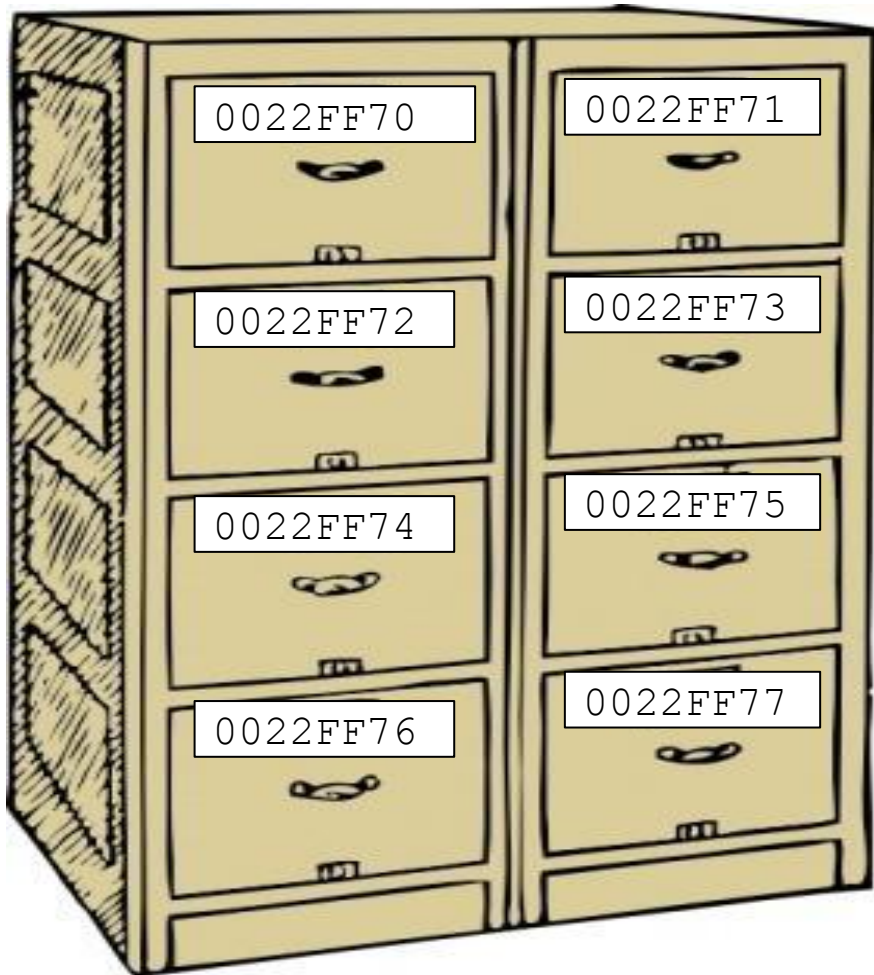
- Suponha que  $y$  armazene o endereço `0022FF74` de uma posição de memória representada pela variável  $x$  e que  $x$  contenha o valor inteiro 3

- Esquematicamente, podemos representar:



- Diz-se que  $y$  é um **ponteiro** para  $x$ , ou que  $y$  aponta para  $x$

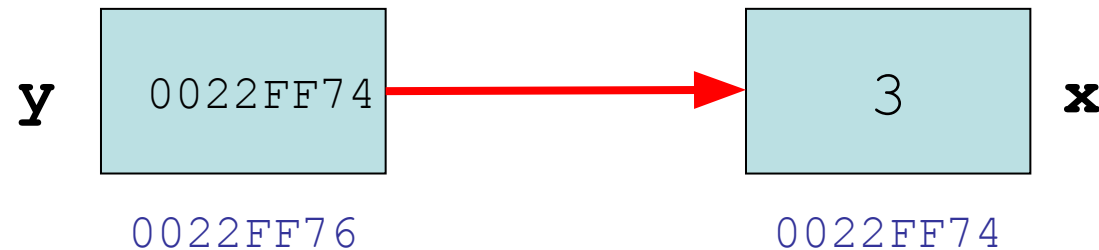
# Endereços de variáveis



Endereço	Variável	Conteúdo
0022FF70	salario	891
0022FF71	c	'a'
0022FF72	idade	8
0022FF73	velocidade	16.1
0022FF74	x	3
0022FF75	km	298347
<b>0022FF76</b>	<b>y</b>	<b>0022FF74</b>

# Endereços de variáveis

- Qual é o tipo da variável  $y$ ?
  - Para **declarar um ponteiro** é preciso saber para qual tipo de valor este ponteiro irá apontar
  - Exemplo do caso anterior:



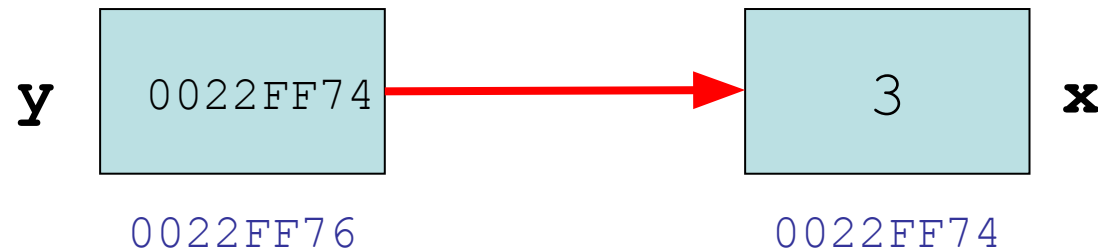
- Neste caso, o ponteiro aponta para um valor inteiro
  - Assim, diz-se que o tipo de  $y$  é `int *`
- A declaração da variável  $y$  será:

`int *y;`

Indica que  $y$  é um ponteiro (para **int**, no caso)

# Endereços de variáveis

- Como acessar o conteúdo do endereço apontado por **y**?



**int \*y;**

Indica que **y** é um ponteiro  
(para **int**, no caso)

- Usa-se o operador **\*** para isso:

```
printf("O conteúdo do endereço apontado por y é: %d", *y);  
//vai imprimir 3
```

# Endereços de variáveis

## ■ Operadores de memória:

- `tipo* var`
  - Declara que `var` armazena endereços com conteúdo do `tipo`
- `&var`
  - Retorna o endereço de memória da variável `var`
- `*var`
  - Acessa o conteúdo da memória em `var`
- `%p`
  - Formata endereços de memória para impressão com `printf`

# Endereços de variáveis

## Operadores de memória:

```
int y = 20;
```

```
int *x = &y;
```

→ Acessar o endereço de memória

x; → Acessar o conteúdo **x = 0x00FFAA**

\*x; → Acessar o conteúdo no endereço de memória ao qual y que aponta **\*x = 20**



# Endereços de variáveis

```
int x = 3;  
int *y;
```

Endereço	Variável	Conteúdo
0022FF70	salario	891
0022FF71	c	'a'
0022FF72	idade	8
0022FF73	velocidade	16.1
0022FF74	x	3
0022FF75	km	298347
0022FF76	y	

# Endereços de variáveis

```
int x = 3;  
int *y;  
y = &x; // y recebe o endereço de x
```

Endereço	Variável	Conteúdo
0022FF70	salario	891
0022FF71	c	'a'
0022FF72	idade	8
0022FF73	velocidade	16.1
0022FF74	x	3
0022FF75	km	298347
0022FF76	y	0022FF74

# Endereços de variáveis

```
int x = 3;  
int *y;  
y = &x; // y recebe o endereço de x  
printf("conteudo de y : %d", *y);  
// *y = conteudo do endereco armazenado em y
```

Endereço	Variável	Conteúdo
0022FF70	salario	891
0022FF71	c	'a'
0022FF72	idade	8
0022FF73	velocidade	16.1
0022FF74	x	3
0022FF75	km	298347
0022FF76	y	0022FF74

**y**

**\*y** ou **\*(0022FF74)**

# Endereços de variáveis

```
int x = 3;
int *y;
y = &x; // y recebe o endereço de x
printf("conteudo de y : %d", *y);
// *y = conteudo do endereco armazenado em y
// usa-se para alterar a variavel apontada
*y = *y + 10;
printf("\n x = %d", x); // imprime 13
```

Endereço	Variável	Conteúdo
0022FF70	salario	891
0022FF71	c	'a'
0022FF72	idade	8
0022FF73	velocidade	16.1
0022FF74	x	3
0022FF75	km	298347
0022FF76	y	0022FF74

# Endereços de variáveis

```
int x = 3;  
int *y;  
y = &x; // y recebe o endereço de x  
printf("conteudo de y : %d", *y);  
// *y = conteudo do endereco armazenado em y  
// usa-se para alterar a variavel apontada  
*y = *y + 10;  
printf("\n x = %d", x); // imprime 13
```

Endereço	Variável	Conteúdo
0022FF70	salario	891
0022FF71	c	'a'
0022FF72	idade	8
0022FF73	velocidade	16.1
0022FF74	x	<b>13</b>
0022FF75	km	298347
0022FF76	y	0022FF74

# Endereços de variáveis

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

```
// gcc -o enderecos enderecos.c && ./enderecos
```

```
int main() {  
    int x = 20;  
    int *y = &x;  
  
    printf("conteudo de x: %d\n", x);  
    printf("endereço de x: %p\n", &x);  
    printf("conteudo de y: %p\n", y);  
    printf("endereço de y: %p\n", &y);  
    printf("conteudo do endereço apontado por y: %d\n", *y);  
    return 0;  
}
```

A screenshot of a terminal window with a dark background. The window has a title bar with a minus sign, a maximize button, and a close button. The command prompt shows the command to compile and run the program: \$ gcc -o enderecos enderecos.c && ./enderecos.

```
$ gcc -o enderecos enderecos.c && ./enderecos
```

- Por que usar **ponteiros**?
  - Permitem manipular a memória do computador manualmente
  - Trabalhar com arquivos, sockets, etc.
  - Passagem de parâmetros via referência:
    - Sem copia! Só passando o endereço de memória...
  - Trabalhar com variáveis fora do escopo no qual foram criadas

# Operadores de incremento e decremento

- Uma operação muito comum em programas de computador é **incrementar de 1** o valor da variável
- Para fazer isso devemos:
  - Somar 1 ao valor atual da variável
  - Armazenar o resultado na própria variável

```
int x = 20;  
x = x + 1; // x = 21
```

- Como a operação **incremento de 1** é muito comum
  - Em C tem-se um operador especial: ++



# Operadores de incremento e decremento

- Ao invés de escrevermos  $x = x + 1$ , podemos escrever:
  - $x++$
- Da mesma forma, para a operação decremento de 1:
  - Em vez de  $x = x - 1$ , podemos escrever:  $x--$
- Os operadores  $++$  e  $--$  podem ser usados como **prefixo** ou como **sufixo** do nome da variável

```
int a = 5;
```

```
int b = 3;
```

```
int c;
```

```
c = a++ + b;
```

```
c = ++a + b;
```



a = 6   b = 3   c = 8



a = 7   b = 3   c = 10

# Operadores de incremento e decremento

- As operações de incremento (++) e decremento (--) são exemplos de **operações combinadas com a atribuição**
- Na linguagem C, sempre que for necessário escrever uma operação de atribuição da forma:  
  
`variavel = variavel operador expressao;`
- Outras operações também podem ser combinadas!

```
x = x + 5;  
x = x - (a + b);  
x = x * (a - b);  
x = x / (x + 1);  
x += 5;  
x -= (a + b);  
x *= (a - b);  
x /= (x + 1);
```

# Operações bit-a-bit

- Tabela-verdade para cada operador:
  - Calculam o resultado lógico (verdadeiro ou falso) de proposições compostas

and (&)

x	y	
	0	1
0	0	0
1	0	1

or (|)

x	y	
	0	1
0	0	1
1	1	1

xor (^)

x	y	
	0	1
0	0	1
1	1	0

# Operações bit-a-bit

- Vou a praia...
  - **x**: se for fim de semana
  - **y**: se fizer sol
- Codificação:
  - 1: sim
  - 0: Não

**and (&)**

<b>x</b>	<b>y</b>	
	0	1
0	0	0
1	0	1

**or (|)**

<b>x</b>	<b>y</b>	
	0	1
0	0	1
1	1	1

**xor (^)**

<b>x</b>	<b>y</b>	
	0	1
0	0	1
1	1	0

# Operações bit-a-bit

- Por uma questão de eficiência, a linguagem C dispõe de operações que **podem ser feitas sobre a representação binária dos números inteiros**

Operador	Operação
<<	deslocamento para a esquerda
>>	deslocamento para a direita
&	conjunção bit-a-bit ( <i>and</i> )
	disjunção bit-a-bit ( <i>or</i> )
^	disjunção exclusiva bit-a-bit ( <i>xor</i> )
~	negação bit-a-bit ( <i>inverso</i> )

# Operações bit-a-bit

## Hexadecimal

## Binário

0FF0

0000 1111 1111 0000

FF00

1111 1111 0000 0000

0FF0 << 4

0FF0 >> 4

0FF0 & FF00

0FF0 | FF00

0FF0 ^ FF00

~ 0FF0

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

```
// gcc -o operacoes_bit operacoes_bit.c && ./operacoes_bit
```

```
int main() {  
    int a = 0x0FF0;  
    int b = 0xFF00;  
    int c;  
  
    c = a << 4;  
    printf("%04X << 4 = %04X\n", a, c);  
    c = a >> 4;  
    printf("%04X >> 4 = %04X\n", a, c);  
    c = a & b;  
    printf("%04X & %04X = %04X\n", a, b, c);  
    return 0;  
}
```



```
$ gcc -o operacoes_bit operacoes_bit.c &&  
./operacoes_bit  
  
0FF0 << 4 = FF00  
0FF0 >> 4 = 00FF  
0FF0 & FF00 = 0F00
```

# Perguntas?

- E-mail:
  - [hector@dcc.ufmg.br](mailto:hector@dcc.ufmg.br)
- Material da disciplina:
  - <https://pedroolmo.github.io/teaching/pdsI.html>
- Github:
  - <https://github.com/h3ct0r>



**Héctor Azpúrua**  
h3ct0r