

## Software Básico

Atividade 2 – Representações de dados. Tamanhos e ordenação.

*Exercício 1.*

Considere o código abaixo do arquivo `ex01_show_bytes_64_32.c` (disponível no Github) `#include <stdio.h>`

```
typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, size_t len) {
    size_t i;
    for (i = 0; i < len; i++)
        printf(" %.2x", start[i]); //line:data:show_bytes_printf
    printf("\n");
}

void show_int(int x) {
    show_bytes((byte_pointer) &x, sizeof(int));
}

void show_long(long x) {
    show_bytes((byte_pointer) &x, sizeof(long));
}

void show_float(float x) {
    show_bytes((byte_pointer) &x, sizeof(float));
}

void show_double(double x) {
    show_bytes((byte_pointer) &x, sizeof(double));
}

void show_pointer(void *x) {
    show_bytes((byte_pointer) &x, sizeof(void *));
}

void test_show_bytes(int val) {
    int ival = val;
    int lval = (long) ival;
    float fval = (float) ival;
```

```

float dval = (double) ival;
int *pval = &ival;
show_int(ival);
show_long(lval);
show_float(fval);
show_double(dval);
show_pointer(pval);
}

int main(int argc, char *argv[]){
    int val = 12345;

    test_show_bytes(val);

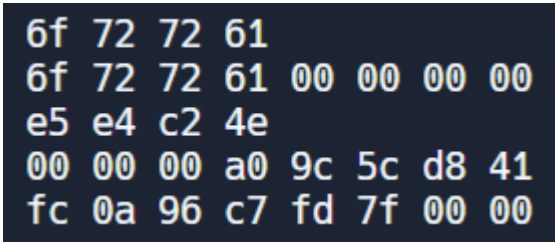
    return 0;
}

```

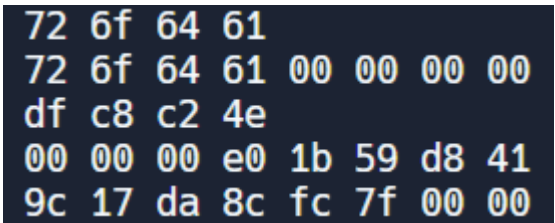
a) Compare com o código ex01\_show\_bytes.c (disponível no Github) apresentado e comentado na sala de aulas. Identifique e comente as principais mudanças no código.

R) A principal mudança no código é que no ex01\_show\_bytes\_64\_32, está adaptado para mostrar valores LongInt para inteiros de tamanho 2147483647, e também pode mostrar valores do tipo double, o que o ex01\_show\_bytes não mostra.

b) Compilar e executar para um tamanho de palavra de 64 bits. Comente o resultado obtido.

A. 

c) Compilar e executar para um tamanho de palavra de 32 bits. Comente o resultado obtido.

A. 

d) Quais as diferenças entre as saídas dos itens (b) e (c). Comente.

### Exercício 2.

Considere as chamadas abaixo da função `show_bytes`:

```
int a = 0x12345678;
byte_pointer ap = (byte_pointer) &a;
show_bytes(ap, 1); /* A. */
show_bytes(ap, 2); /* B. */
show_bytes(ap, 3); /* C. */
```

Indique os valores que serão impressos em cada linha se função for chamada em uma máquina little endian ou em uma máquina big-endian

Chamada	Little-Endian	Big-Endian
A	78	12
B	78 56	12 34
C	78 56 34	12 34 56

### Exercício 3.

Que será impresso como resultado da chamada a função `show_bytes` considerando o código a

```
const char *m = "mnopqr";
seguir show_bytes((byte_pointer) m, strlen(m));
```

6d 6e 6f 70 71 72

### Exercício 4.

Leia o texto a seguir sobre o standard Unicode para codificação de arquivos

texto *The Unicode standard for text encoding*

The ASCII character set is suitable for encoding English-language documents, but it does not have much in the way of special characters, such as the French ‘,c’. It is wholly unsuited for encoding documents in languages such as Greek, Russian, and Chinese. Over the years, a variety of methods have been developed to encode text for different languages. The Unicode Consortium has devised the most comprehensive and widely accepted standard for encoding text. The current Unicode standard (version 7.0) has a repertoire of over 100,000 characters supporting a wide range of languages, including the ancient languages of Egypt and Babylon. To their credit, the Unicode Technical Committee rejected a proposal to include a standard writing for Klingon, a fictional civilization from the television series Star Trek.

The base encoding, known as the “Universal Character Set” of Unicode, uses a 32-bit representation of characters. This would seem to require every string of text to consist of 4 bytes per character. However, alternative codings are possible where common characters require just 1 or 2 bytes, while less common ones require more. In particular, the UTF-8 representation encodes each character as a sequence of bytes, such that the standard ASCII characters use the same single byte encodings as they have in ASCII, implying that all ASCII byte sequences have the same meaning in UTF-8 as they do in ASCII.

The Java programming language uses Unicode in its representations of strings. Program libraries are also available for C to support Unicode.

#### Exercício 5.

Complete a tabela abaixo com os resultados de avaliar operações booleanas sobre vetores de

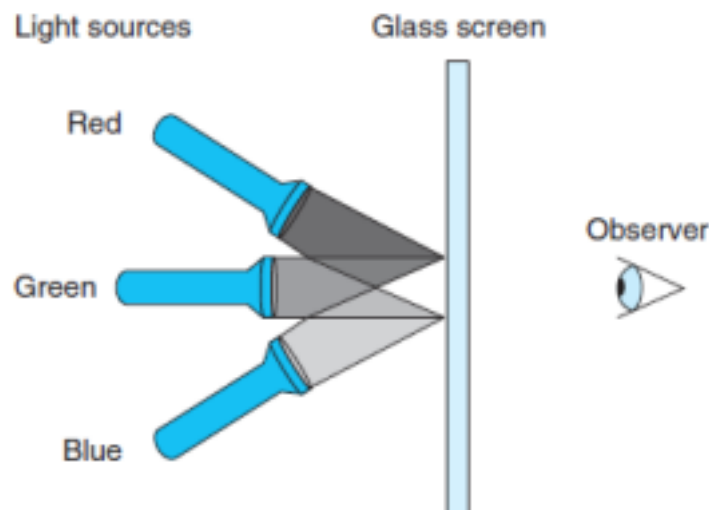
Operation	Result
$a$	[01001110]
$b$	[11100001]
$\sim a$	_____
$\sim b$	_____
$a \& b$	_____
$a   b$	_____
$a \wedge b$	_____

bits

Operation	Result
$a$	[01001110]
$b$	[11100001]
$\sim a$	[10110001]
$\sim b$	[00011110]
$a \& b$	[01000000]
$a   b$	[11101111]
$a \wedge b$	[10101111]

### Exercício 6.

Os computadores geram imagens coloridas em uma tela de vídeo misturando três cores diferentes de luz: vermelho, verde e azul. Imagine um esquema simples, com três luzes diferentes, cada uma podendo ser ligada ou desligada, projetando-se em uma tela de vidro:



Podemos então criar oito cores diferentes com base na ausência (0) ou presença (1) das fontes de luz R, G e B.

<i>R</i>	<i>G</i>	<i>B</i>	Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

Cada uma dessas cores pode ser representada como um vetor de bits de comprimento 3 e podemos aplicar operações booleanas a elas.

a) O complemento de uma cor é formado apagando as luzes que estão acesas e acendendo as que estão apagadas. Qual seria o complemento de cada uma das oito cores listadas acima?

- **(Black):** 000 -> Complemento: 111 (Branco)
- **(Blue):** 001 -> Complemento: 110 (Amarelo)
- **(Green):** 010 -> Complemento: 101 (Magenta)
- **(Cyan):** 011 -> Complemento: 100 (Vermelho)
- **(Red):** 100 -> Complemento: 011 (Ciano)
- **(Magenta):** 101 -> Complemento: 010 (Verde)

- **(Yellow):** 110 -> Complemento: 001 (Azul)
- **(White):** 111 -> Complemento: 000 (Preto)

b) Obtenha o resultado de aplicar operações booleanas nas cores a seguir. Comente os resultados.

Blue | Green = \_\_\_\_\_  
 Yellow & Cyan = \_\_\_\_\_  
 Red ^ Magenta = \_\_\_\_\_

$$B) 001 | 010 = 011$$

$$110 \& 011 = 010$$

$$100 \wedge 101 = 001$$

Exercício 7.

Leia o texto a seguir que descreve propriedades do álgebra booleana.

#### Web Aside DATA:BOOL More on Boolean algebra and Boolean rings

The Boolean operations  $|$ ,  $\&$ , and  $\sim$  operating on bit vectors of length  $w$  form a *Boolean algebra*, for any integer  $w > 0$ . The simplest is the case where  $w = 1$  and there are just two elements, but for the more general case there are  $2^w$  bit vectors of length  $w$ . Boolean algebra has many of the same properties as arithmetic over integers. For example, just as multiplication distributes over addition, written  $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ , Boolean operation  $\&$  distributes over  $|$ , written  $a \& (b | c) = (a \& b) | (a \& c)$ . In addition, however, Boolean operation  $|$  distributes over  $\&$ , and so we can write  $a | (b \& c) = (a | b) \& (a | c)$ , whereas we cannot say that  $a + (b \cdot c) = (a + b) \cdot (a + c)$  holds for all integers.

When we consider operations  $\sim$ ,  $\&$ , and  $\sim$  operating on bit vectors of length  $w$ , we get a different mathematical form, known as a *Boolean ring*. Boolean rings have many properties in common with integer arithmetic. For example, one property of integer arithmetic is that every value  $x$  has an *additive inverse*  $-x$ , such that  $x + -x = 0$ . A similar property holds for Boolean rings, where  $\sim$  is the “addition” operation, but in this case each element is its own additive inverse. That is,  $a \sim a = 0$  for any value  $a$ , where we use 0 here to represent a bit vector of all zeros. We can see this holds for single bits, since  $0 \sim 0 = 1 \sim 1 = 0$ , and it extends to bit vectors as well. This property holds even when we rearrange terms and combine them in a different order, and so  $(a \sim b) \sim a = b$ . This property leads to some interesting results and clever tricks, as we will explore in Problem 2.10.

Exercício 8.

Considerando a função inplace\_swap mostrada em sala de aulas, foi decidido criar uma função que inverta os elementos de um vetor trocando os elementos das extremidades opostas, isto é primeiro troca com último, segundo com penúltimo e continuar até alcançar o centro do vetor, conforme a figura abaixo

1	2	3	4	5
---	---	---	---	---



6	5	4	3	2	1
---	---	---	---	---	---

A equipe de desenvolvimento produziu o código abaixo:

```
void reverse_array(int a[], int cnt) {  
    int first, last;  
    for (first = 0, last = cnt-1;  
        first <= last;  
        first++,last--)  
        inplace_swap(&a[first], &a[last]);  
}
```

a) Qual seria o resultado de executarmos a função com as entradas:

a[] = {1, 2, 3, 4, 5, 6} cnt=6

a[] = {1, 2, 3, 4} cnt=4

<b>Input:</b> x= 01 0000 0001 y= 06 0000 0110 <b>Step 1:</b> x= 01 0000 0001 y= 07 0000 0111 <b>Step 2:</b> x= 06 0000 0110 y= 07 0000 0111 <b>Step 3:</b> x= 06 0000 0110 y= 01 0000 0001	<b>Input:</b> x= 02 0000 0010 y= 05 0000 0101 <b>Step 1:</b> x= 02 0000 0010 y= 07 0000 0111 <b>Step 2:</b> x= 05 0000 0101 y= 07 0000 0111 <b>Step 3:</b> x= 05 0000 0101 y= 02 0000 0010	<b>Input:</b> x= 03 0000 0011 y= 04 0000 0100 <b>Step 1:</b> x= 03 0000 0011 y= 07 0000 0111 <b>Step 2:</b> x= 04 0000 0100 y= 07 0000 0111 <b>Step 3:</b> x= 04 0000 0100 y= 03 0000 0011	<b>Input:</b> x= 01 0000 0001 y= 04 0000 0100 <b>Step 1:</b> x= 01 0000 0001 y= 05 0000 0101 <b>Step 2:</b> x= 04 0000 0100 y= 05 0000 0101 <b>Step 3:</b> x= 04 0000 0100 y= 01 0000 0001	<b>Input:</b> x= 02 0000 0010 y= 03 0000 0011 <b>Step 1:</b> x= 02 0000 0010 y= 01 0000 0001 <b>Step 2:</b> x= 03 0000 0011 y= 01 0000 0001 <b>Step 3:</b> x= 03 0000 0011 y= 02 0000 0010
---	---	---	---	---

A.

b) Qual seria o resultado de executarmos a função com as entradas:

a[] = {1, 2, 3, 4, 5} cnt=5

a[] = {1, 2, 3, 4, 5, 6, 7} cnt=7

Input: x= 02 0000 0010 y= 06 0000 0110 Step 1: x= 02 0000 0010 y= 04 0000 0100 Step 2: x= 06 0000 0110 y= 04 0000 0100 Step 3: x= 06 0000 0110 y= 02 0000 0010	Input: x= 01 0000 0001 y= 05 0000 0101 Step 1: x= 01 0000 0001 y= 04 0000 0100 Step 2: x= 05 0000 0101 y= 04 0000 0100 Step 3: x= 05 0000 0101 y= 01 0000 0001	Input: x= 02 0000 0010 y= 04 0000 0100 Step 1: x= 02 0000 0010 y= 06 0000 0110 Step 2: x= 04 0000 0100 y= 06 0000 0110 Step 3: x= 04 0000 0100 y= 02 0000 0010	Input: x= 03 0000 0011 y= 03 0000 0011 Step 1: x= 00 0000 0000 y= 00 0000 0000 Step 2: x= 00 0000 0000 y= 00 0000 0000 Step 3: x= 00 0000 0000 y= 00 0000 0000	Input: x= 01 0000 0001 y= 07 0000 0111 Step 1: x= 01 0000 0001 y= 06 0000 0110 Step 2: x= 07 0000 0111 y= 06 0000 0110 Step 3: x= 07 0000 0111 y= 01 0000 0001
---	---	---	---	---



<pre> Input: x= 03 0000 0011 y= 05 0000 0101 Step 1: x= 03 0000 0011 y= 06 0000 0110 Step 2: x= 05 0000 0101 y= 06 0000 0110 Step 3: x= 05 0000 0101 y= 03 0000 0011 </pre>	<pre> Input: x= 04 0000 0100 y= 04 0000 0100 Step 1: x= 00 0000 0000 y= 00 0000 0000 Step 2: x= 00 0000 0000 y= 00 0000 0000 Step 3: x= 00 0000 0000 y= 00 0000 0000 </pre>
---	---

- c) Para quais comprimentos do vetor a função opera corretamente. Para quais oferece um resultado incorreto?
- d) Quais os valores das variáveis first e last na última iteração para as chamadas à função reverse\_array com vetores de comprimento [ímpar (item b)]? Por que está chamada à função inplace\_swap define um elemento do vetor como zero?
- e) Que aprimoramento devemos efetuar no código da função para eliminar este problema?

#### Exercício 9.

Considerando as operações *bitwise*, crie máscaras e escreva expressões em C em termos da variável *x* para obter como resultados os seguintes valores, a modo de referência mostramos o resultado esperado considerando *x = 0x87654321*

- (a) O byte menos significativo de *x* permanece inalterado, com todos os outros bits definidos como 0. [0x00000021]  
`result = x & 0x000000FF`
- (b) O byte mais significativo de *x* permanece inalterado, com todos os outros bits definidos como 0.

[0x87000000]

result = x & 0xFF000000;

(c) Calcula-se o complemento de todos os bytes, exceto o byte menos significativo de x que permanece inalterado. [0x789ABC21]

result = (x & 0x000000FF) | (~x & 0xFFFFF00);

(d) O byte menos significativo é definido como todos os bits iguais a 1 e todos os outros bytes de x permanecem inalterados. [0x876543FF]

result = (x & 0xFFFFF00) | 0x000000FF;

### Exercício 10

Considerando que a = 0xA55C e b = 0xE146, complete a tabela abaixo indicando os valores em bytes das expressões em C

Expression	Value	Expression	Value
a & b	_____	a && b	_____
a   b	_____	a    b	_____
~a   ~b	_____	!a    !b	_____
a & !b	_____	a && ~b	_____

a&b = 0xA144		a && b = 0x0001
a   b = 0xE55E		a    b = 0x0001
~a   ~b = FFFF5EBB		!a    !b = 0x0000
a & !b = 0000		a && ~b = 0x0001

### Exercício 11\*

Usando apenas operações lógicas e de nível de bit, escreva uma expressão C que seja equivalente a  $x == y$ . Em outras palavras, retornará 1 quando x e y forem iguais e 0 caso contrário.

isEqual = !(x ^ y)

isEqual vai ser 1 se forem iguais e 0 se forem diferentes.

### Exercício 12

Complete a tabela abaixo com os resultados das operações de deslocamento de bits em dados armazenados em um único byte.

HEX - A	BINARY - A
0xFA	1111 1010
0x64	0110 0100
0x72	0111 0010
0x44	0100 0100

BINARY - A << 2	HEX - A << 2
1110 1000	0xE8
1001 0000	0x90
1100 1000	0xC8
0001 0000	0x10

BINARY - A >> 3 (LOGICAL)	HEX - A >> 3 (LOGICAL)
0001 1111	0x1F
0000 1100	0xC
0000 1110	0xE
0000 1000	0x8

BINARY - A >> 3 (ARITHMETIC)	HEX - A >> 3 (ARITHMETIC)
1111 1111	0xFF
1110 1100	0xEC
1110 1110	0xEE
1110 1000	0xE8

### Exercício 13

Leia o texto abaixo sobre operações de deslocamento com  $k > w$ , isto é o valor do deslocamento é maior que o tamanho da palavra.

#### Aside Shifting by $k$ , for large values of $k$

For a data type consisting of  $w$  bits, what should be the effect of shifting by some value  $k \geq w$ ? For example, what should be the effect of computing the following expressions, assuming data type `int` has  $w = 32$ :

```
int    lval = 0xFEDCBA98 << 32;
int    aval = 0xFEDCBA98 >> 36;
unsigned uval = 0xFEDCBA98u >> 40;
```

The C standards carefully avoid stating what should be done in such a case. On many machines, the shift instructions consider only the lower  $\log_2 w$  bits of the shift amount when shifting a  $w$ -bit value, and so the shift amount is computed as  $k \bmod w$ . For example, with  $w = 32$ , the above three shifts would be computed as if they were by amounts 0, 4, and 8, respectively, giving results

```
lval    0xFEDCBA98
aval    0xFFEDCBA9
uval    0x00FEDCBA
```

This behavior is not guaranteed for C programs, however, and so shift amounts should be kept less than the word size.

Java, on the other hand, specifically requires that shift amounts should be computed in the modular fashion we have shown.

Modifique o exemplo oferecido em sala de aulas para verificar o funcionamento deste deslocamento na implementação de linguagem C disponível em seu computador.

### Exercício 14

Faça uma pesquisa estabelecendo a precedência de operadores em linguagem C quando operações bit-a-bit e de deslocamento aparecem na mesma instrução que operações aritméticas. Leia o quadro abaixo sob a precedência em operações de deslocamento.

#### Aside Operator precedence issues with shift operations

It might be tempting to write the expression `1<<2 + 3<<4`, intending it to mean `(1<<2) + (3<<4)`. However, in C the former expression is equivalent to `1 << (2+3) << 4`, since addition (and subtraction) have higher precedence than shifts. The left-to-right associativity rule then causes this to be parenthesized as `(1 << (2+3)) << 4`, giving value 512, rather than the intended 52.

Getting the precedence wrong in C expressions is a common source of program errors, and often these are difficult to spot by inspection. When in doubt, put in parentheses!