ARQCP

Apontadores em C

aula TP

2024/2025

André Andrade

lao@isep.ipp.pt

Aviso

- A documentação oficial de ARQCP são os slides disponibilizados no Moodle do ISEP de ARQCP

- Embora estes slides sejam criados com base na informação oficial podem existir algumas diferenças

- Aconselha-se a consulta da informação oficial

Introdução

- Um programa interpreta a memória como sendo um grande *array,* onde cada elemento desse *array* é um Byte
- Não se utiliza o termo "índice" quando se pretende referir uma posição de memória, usa-se o termo **endereço**
  - Em cada **endereço** encontra-se armazenado **um Byte**
- Quando se declara uma variável, está-se a reservar um bloco de memória (um ou mais Bytes linearmente contínuos em memória)
- Cada variável tem o seu próprio **endereço**
  - **Endereço** de uma variável é a posição (no *array* de memória) a partir de onde a informação da variável está armazenada

Introdução

- Em IA-32, os endereços são compostos por 32 bits
  - Permite gerar 2^{32} = 4 294 967 296 endereços distintos
  - Permite endereçar até 4GiB de memória RAM
- Em x86-64, os endereços são compostos por 64 bits
  - Permite gerar 2^{64} = 18 446 744 073 709 551 616 endereços distintos
  - Permite endereçar até 16EiB = 1 048 576 TiB = 1 073 741 824 GiB de memória RAM

- Atualmente os CPUs x86-64 usam *address lines* de 48 bits
  - Teoricamente, podem endereçar até 256TiB de memória RAM
- Pode consultar a informação do seu sistema com o comando Linux: **`lscpu`**

Apontadores

- O **C** tem variáveis especiais denominadas de apontadores
  - Armazenam endereços de memória
- Os apontadores são declarados como as variáveis normais, com o tipo que lhes está associado
- Os apontadores têm sempre o mesmo tamanho
  - O tamanho do endereço da arquitetura subjacente
    - 64 bits = 8 Bytes em x86-64
    - 32 bits = 4 Bytes em IA-32
- Os apontadores permitem acesso direto à memória
  - É possível modificar os valores armazenados nas posições de memória dos apontadores
- Algumas tarefas são mais simples/mais eficientes de implementar quando se usam apontadores
- Outras, apenas são possíveis se usarmos apontadores (p. ex., alocação de memória dinâmica)

Apontadores: Utilização e notação

- Tal como qualquer outra variável em **C**, um apontador necessita ser declarado antes de ser usado

- Na declaração de um apontador, coloca-se o tipo de dados para o qual o apontador aponta, um '*****' e o nome da variável. Exemplos:

```
int *ptr1; /* declara um apontador para um int */
char *ptr2; /* declara um apontador para um char */
```

- Para obter o endereço de uma variável, utiliza-se '**&**' antes do seu identificador. Exemplos:

```
int x;
char c;
ptr1 = &x; /* atribui a ptr1, o endereço onde x está armazenado */
ptr2 = &c; /* atribui a ptr2, o endereço onde c está armazenado */
```

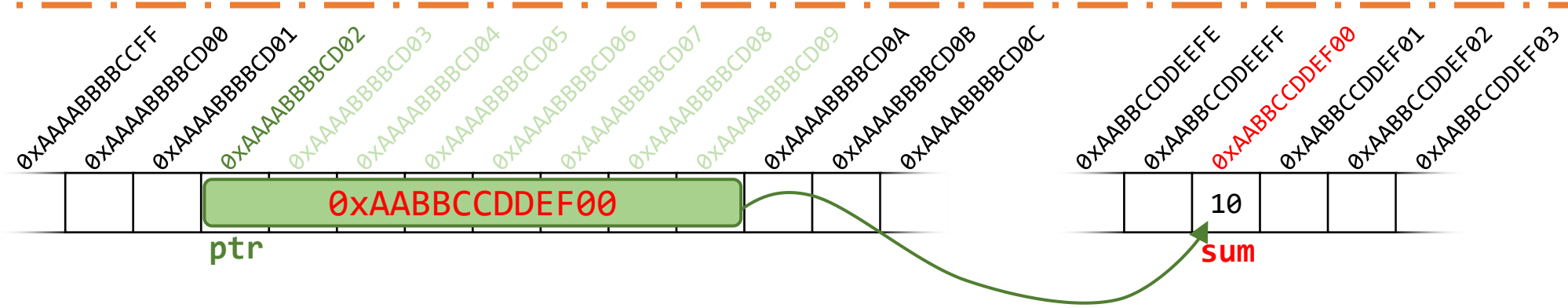
- O operador *dereference* '*****' acede ao valor que está armazenado num endereço de memória. Exemplos:

```
*ptr1 = 10; /* atribui 10 à variável para a qual ptr1 aponta (resulta no mesmo que x=10) */
*ptr2 = 'A'; /* atribui 'A' à variável para a qual ptr2 aponta (resulta no mesmo que c='A') */
```

Conteúdo e endereço de uma variável

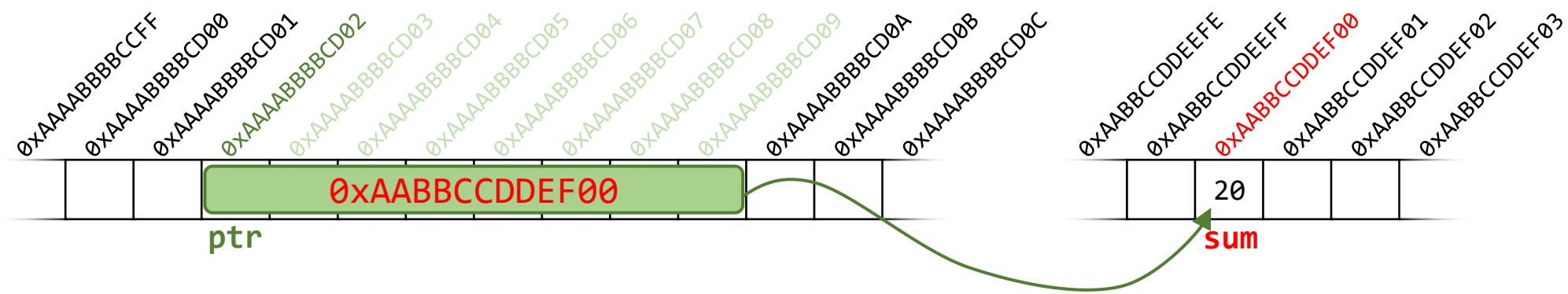
```
char sum;
char *ptr;
sum = 10;
```


```
ptr = &sum;
```



Para nos referirmos ao conteúdo de um endereço de memória usámos o operador *dereference*

```
*ptr = 20;
```



Notas importantes

- Os apontadores devem ser sempre inicializados para um endereço válido antes de serem usados
  - Utilizar um apontador não inicializado resulta em comportamento indefinido!

- Modo errado:

```
int x;
int *ptr; /* apontador não inicializado */
*ptr=22; /* muito má ideia!!! */
```

Este código, muito provavelmente, resulta em *segmentation fault*.

Segmentation fault ocorre quando um programa tenta aceder a um endereço de memória inválido. O sistema operativo deteta o acesso e termina o programa.

- Modo correto:

```
int x;
int *ptr;
ptr=&x; /* agora o apontador está inicializado com o endereço de x */
*ptr=22; /* atribuir 22 à memória apontada por ptr, ou seja, x */
```

Big/Little Endian

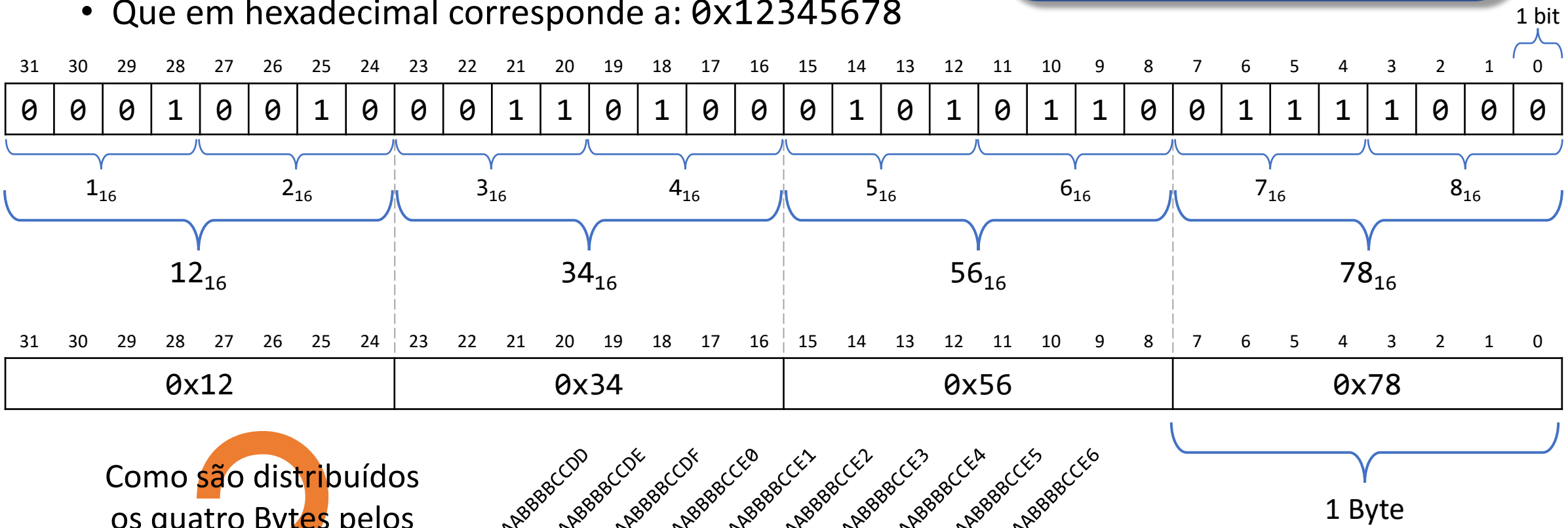
- Foi referido que a memória é um grande *array* de Bytes…
  - E como se armazena informação maior que 1 Byte?
  - Fácil: particiona-se a informação em blocos de 1 Byte e armazena-se!
- O que origina duas formas de armazenamento em memória:
  - **Big Endian**
    - Armazena o Byte mais significativo no menor dos endereços
    - Adotado em plataformas da Sun, PPC Mac, transmissão de informação na Internet
  - **Little Endian**
    - Armazena o Byte menos significativo no menor dos endereços
    - Adotado por x86, ARM

Exemplo:

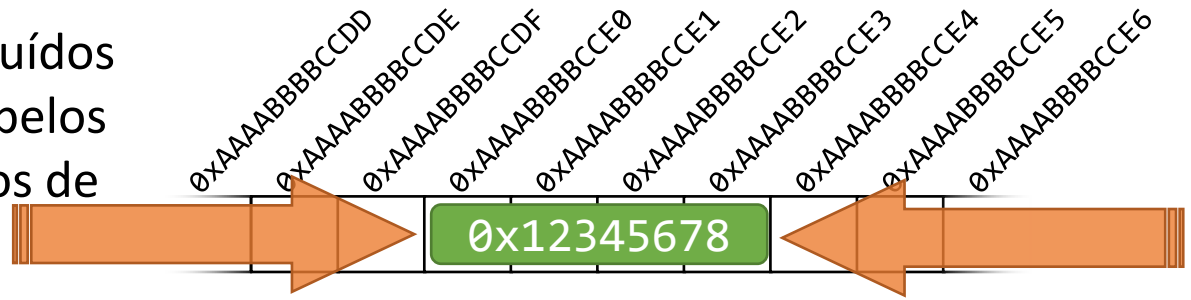
Relembrar:

- O prefixo "0x" indica que a representação do número é hexadecimal
- Um grupo de 4 bits pode ser escrito com um único símbolo em hexadecimal
- Assim, um Byte (8 bits) pode ser escrito com apenas dois símbolos hexadecimais

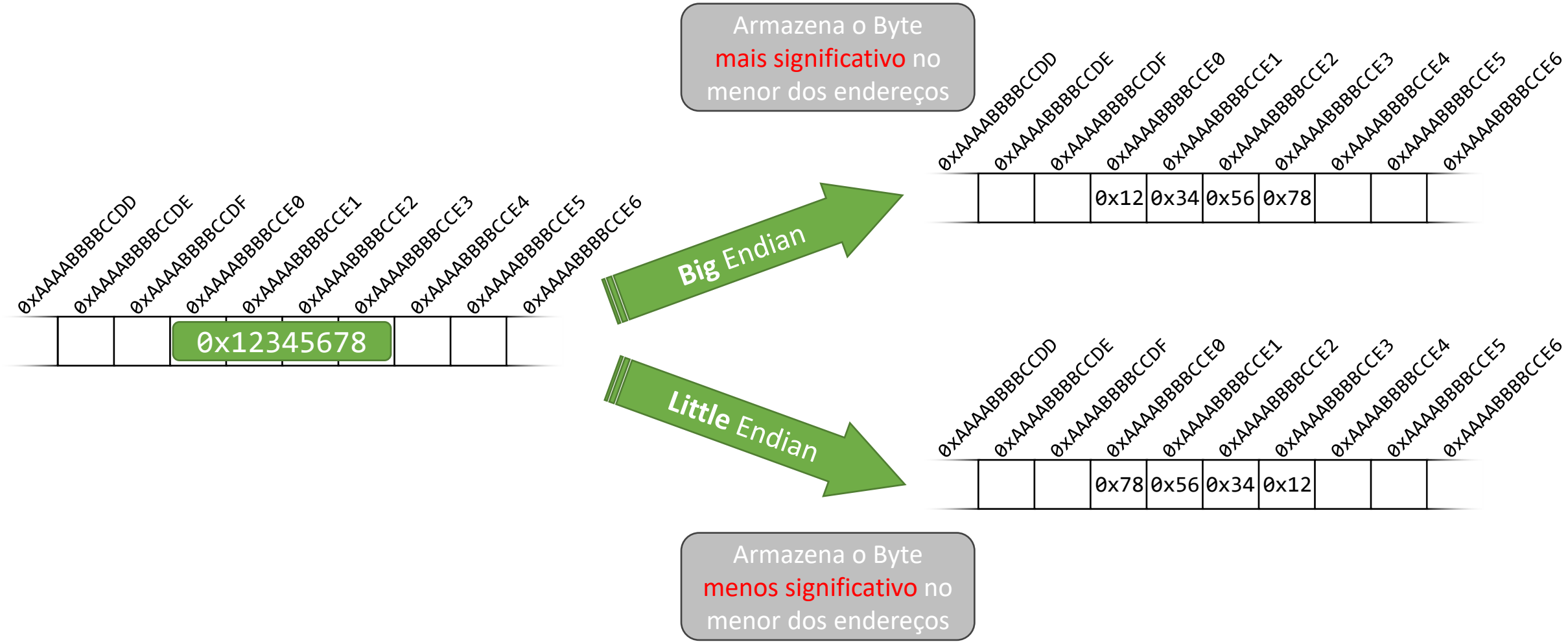
- Armazenar o número **int**: 305419896
  - Que em hexadecimal corresponde a: 0x12345678



Como são distribuídos os quatro Bytes pelos quatro endereços de memória?



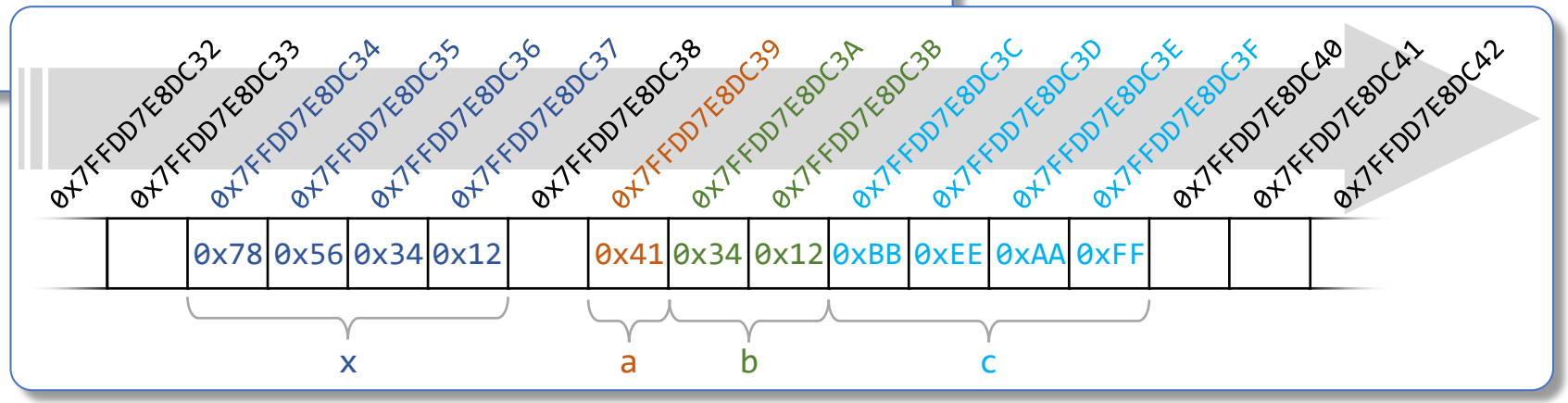
Exemplo: Little Endian vs Big Endian



Representação de informação em x86-64 (Little Endian)

```c
#include <stdio.h>

int main(void) {
    unsigned int x = 0x12345678;
    char a = 'A';
    short b = 0x1234;
    unsigned int c = 0xFFAAEEBB;

    /* %p é um formatador que imprime o endereço do apontador */
    printf("endereço onde x está armazenado: %p\n", &x);
    printf("endereço onde a está armazenado: %p\n", &a);
    printf("endereço onde b está armazenado: %p\n", &b);
    printf("endereço onde c está armazenado: %p\n", &c);

    return 0;
}
```

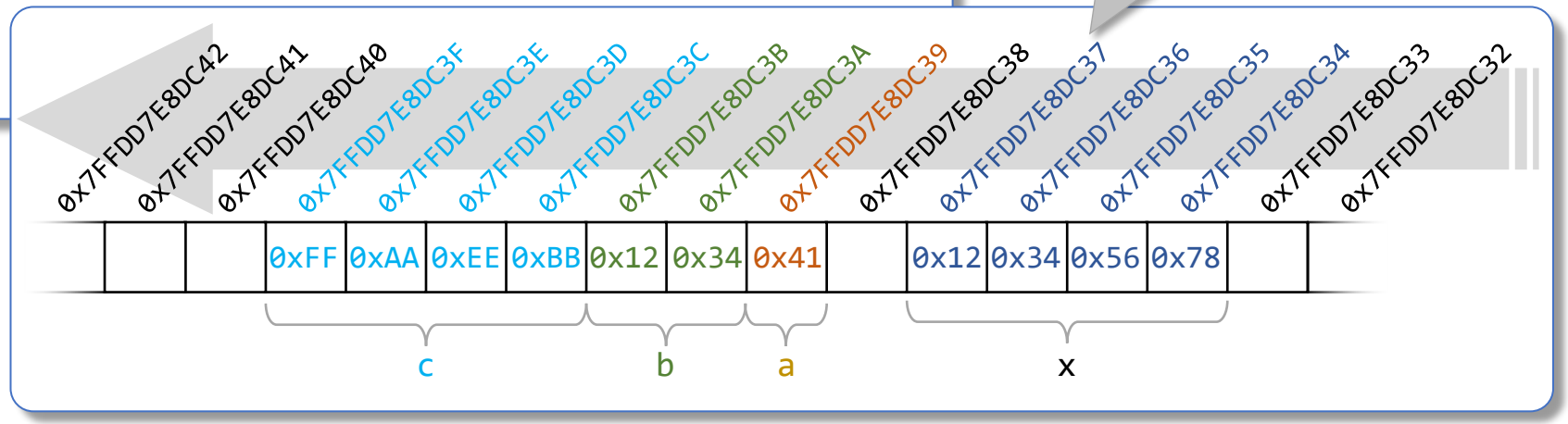
Dec	Hx	Oct	Html	Chr
64	40	100	@	@
65	41	101	A	A
66	42	102	B	B
67	43	103	C	C
68	44	104	D	D
69	45	105	E	E
70	46	106	F	F
71	47	107	G	G
72	48	110	H	H
73	49	111	I	I
74	4A	112	J	J



Representação de informação em x86-64 (Little Endian)

```c
#include <stdio.h>

int main(void) {
    unsigned int x = 0x12345678;
    char a = 'A';
    short b = 0x1234;
    unsigned int c = 0xFFAAEEBB;

    /* %p é um formatador que imprime o endereço do apontador */
    printf("endereço onde x está armazenado: %p\n", &x);
    printf("endereço onde a está armazenado: %p\n", &a);
    printf("endereço onde b está armazenado: %p\n", &b);
    printf("endereço onde c está armazenado: %p\n", &c);

    return 0;
}
```

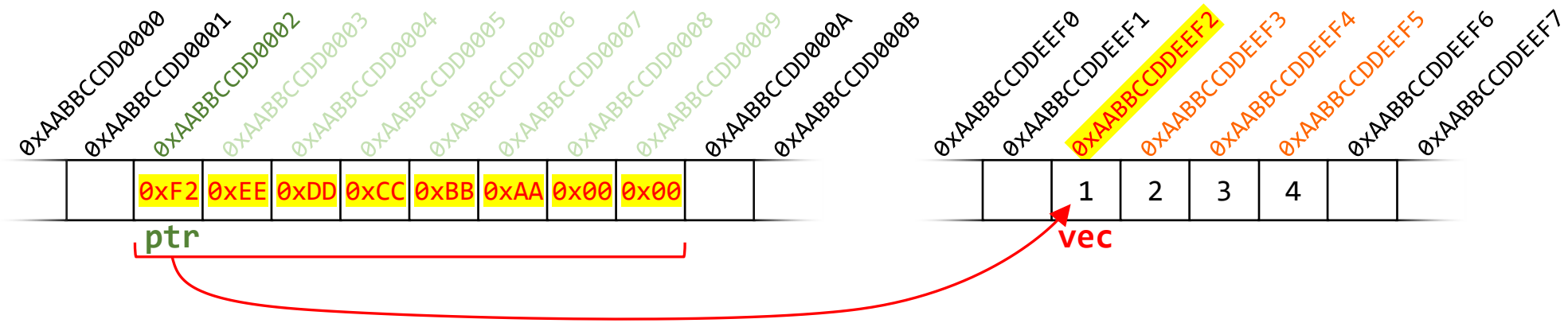
Sugestão:
Quando fizer confusão interpretar os números invertidos, podemos sempre representar os endereços de memória no sentido oposto…



Arrays

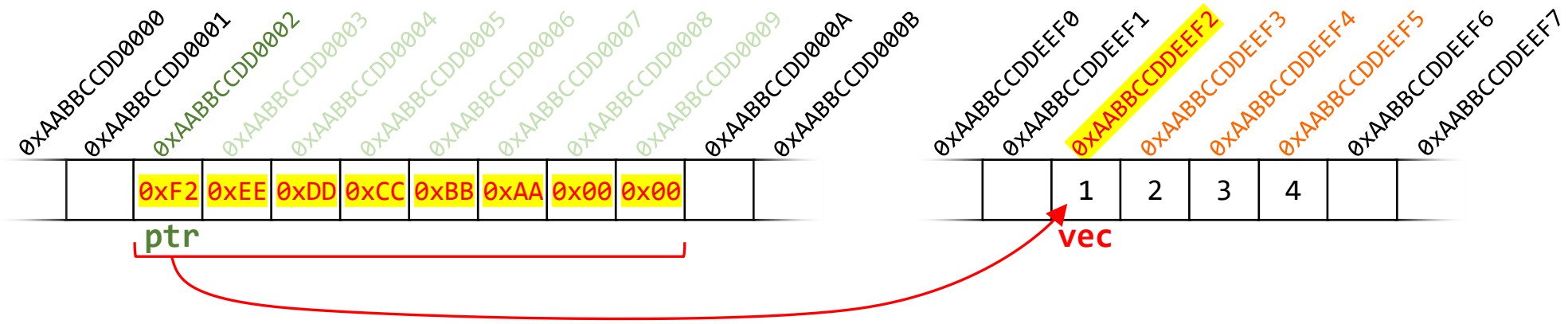
```
char vec[] = {1,2,3,4};
char *ptr = NULL;
```


```
ptr = vec; /* nos arrays, o identificador é o endereço; portanto não há necessidade do '&' */
```

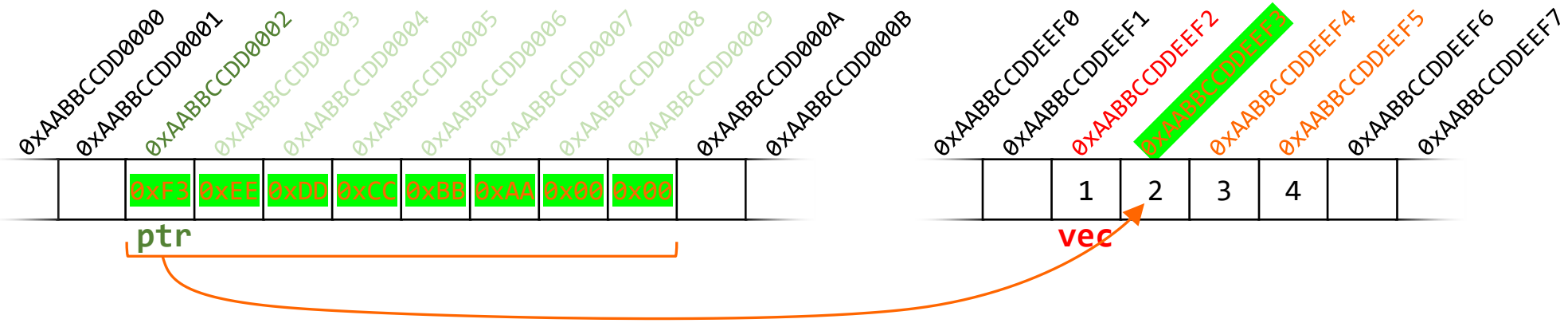


Aritmética de apontadores

Do slide anterior:



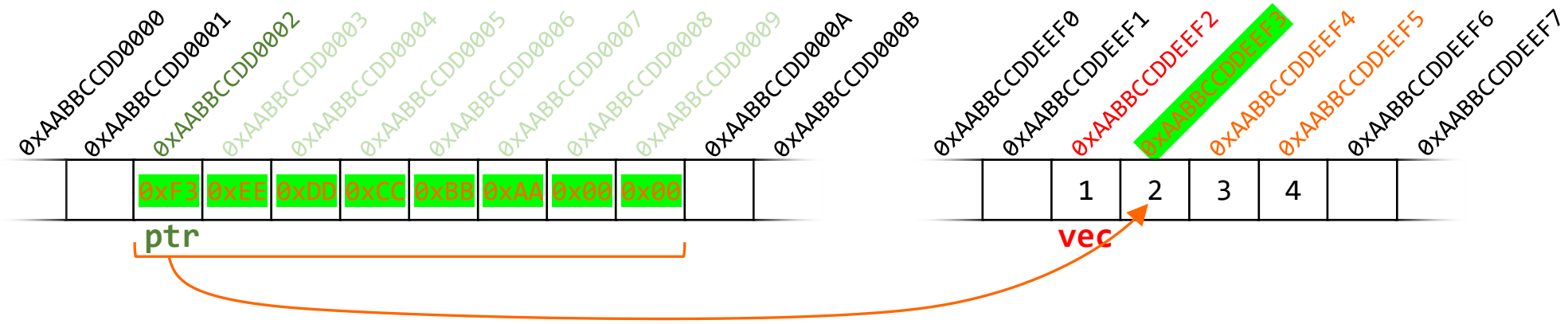

```
++ptr; /* ptr agora aponta para o segundo element de vec */
```



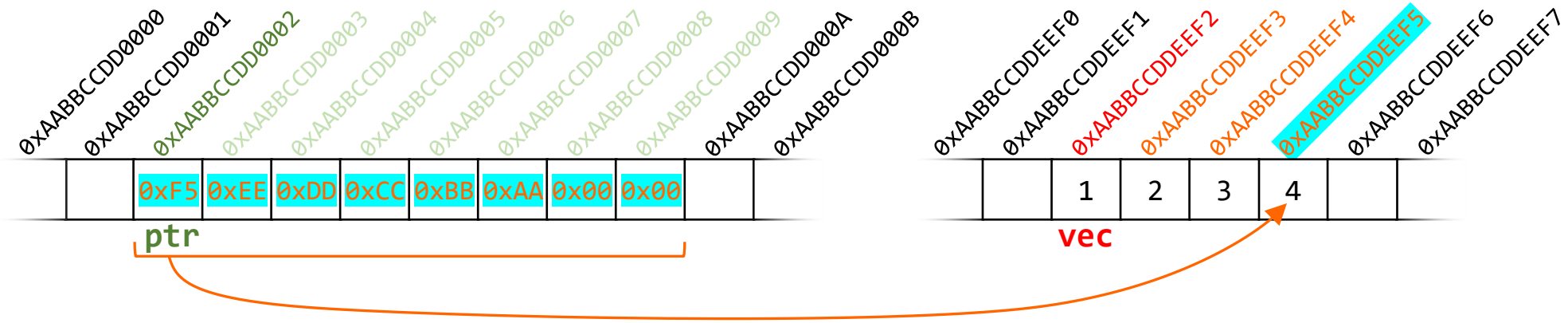

```
printf("%hhd\n", *ptr); /* imprime: 2 */
```

Aritmética de apontadores

Do slide anterior:




```
ptr += 2; /* ptr agora aponta para o quarto element de vec */
```




```
printf("%hhd\n", *ptr); /* imprime: 4 */
```

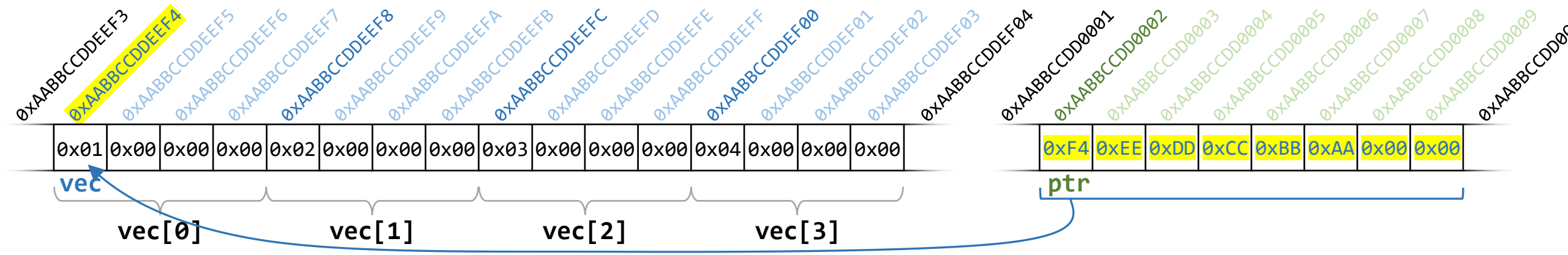
Aritmética de apontadores

- Em **C**, o tipo de apontador é relevante quando se aplica aritmética de apontadores
  - O compilador decide quanto deve somar ou subtrair com base no tipo do apontador

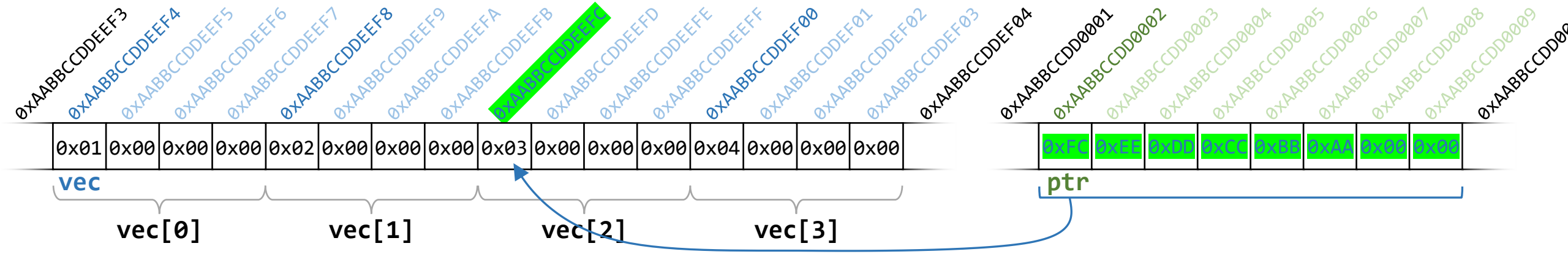
```
int *ptr_total; /* Em x86-64 avança/recua 4 endereços por cada incremento/decremento */
char *ptr_name; /* Em x86-64 avança/recua 1 endereço por cada incremento/decremento */
```

Exemplo1: Aritmética de apontadores

```
int vec[] = {1, 2, 3, 4};
int *ptr = vec;
```

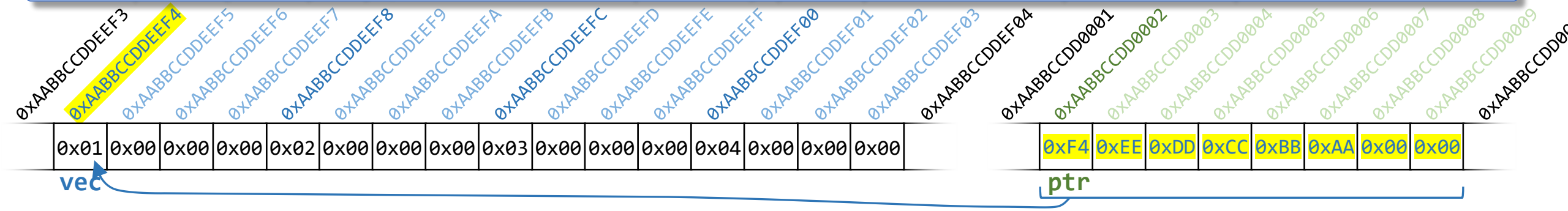



```
ptr = ptr + 2;
```

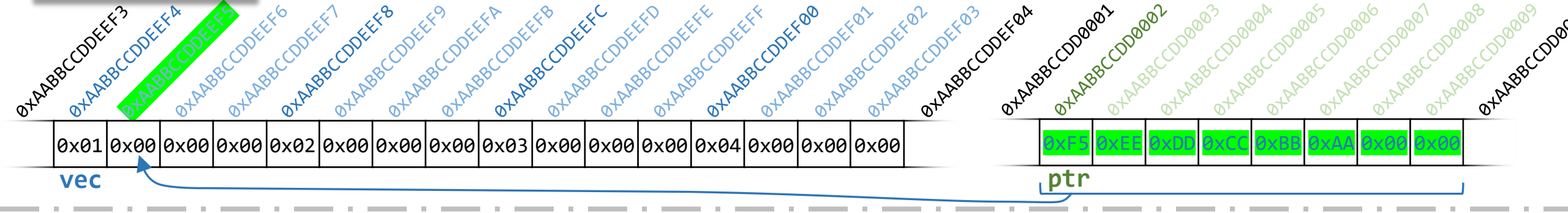


Exemplo2: Aritmética de apontadores

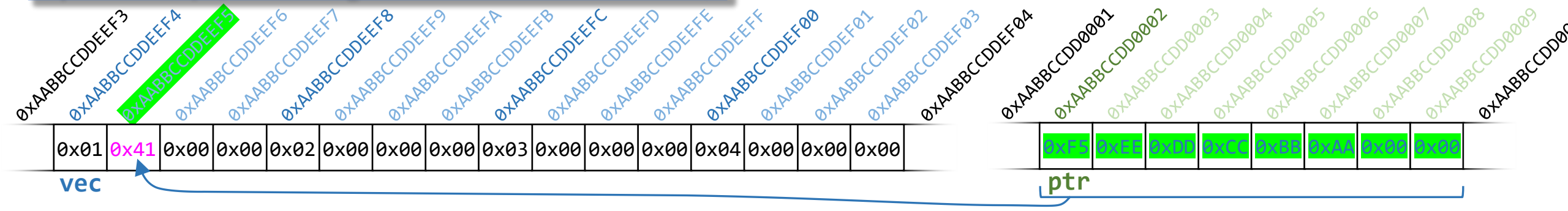
```
int vec[] = {1, 2, 3, 4};
char *ptr = (char*)vec; /* Converter um apontador, para apontador de tipo distinto precisa de cast */
```




```
ptr = ptr + 1;
```




```
*ptr = 'A'; /* O código  ASCII de 'A' é 0x41 */
```



Aritmética de apontadores

```c
#include <stdio.h>

int main(void) {
    char vec[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    const int VEC_SIZE = sizeof(vec)/sizeof(vec[0]);

    short *ptr = (short*)vec; /* é necessário cast para evitar warning... */
    int i=0;

    for(i=0; i<VEC_SIZE/2; ++i) {
        vec[i] = *(ptr + i);
        printf("%hhd ", vec[i]);
    }
    printf("\n");

    return 0;
}
```

Qual será o *output* do código?

a) 1 2 3 4 5
b) 2 3 4 5 6
c) 1 3 5 7 9 ✓
d) Nenhuma das opções

Qual o conteúdo de vec no final?

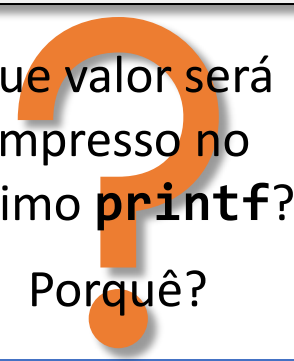
a) {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
b) {1, 3, 5, 7, 9, 6, 7, 8, 9, 10} ✓
c) {513, 1027, 1541, 2055, 2569, 6, 7, 8, 9, 10}
d) Nenhuma das opções

Console I/O: Output

- Impressão na linha de comandos usando as funções `printf` e `puts`

```c
#include <stdio.h>

int main(void) {
   int a=2, b=3;
   int *ptr1=&a, *ptr2=&b;

   printf("Value: %d\n", *ptr1); /* imprime o valor 2 */

   printf("Address: %p (%p)\n", &a, ptr1); /* imprime o endereço de a */

   if(*ptr1 == *ptr2) puts("Same Value"); /* false */

   *ptr1 = 3;
   if(*ptr1 == *ptr2) puts("Now same Value"); /* true */

   ptr1 = ptr2;
   if(ptr1 == ptr2) puts("Now same address"); /* true */

   prinft("a = %d\n ", a); /* true */

   return 0;
}
```

A função **puts** escreve no *stdout* a *string* que recebe seguida de *newline*



Que valor será impresso no último **printf**?

Porquê?

Console I/O: Input

- Receber dados do utilizador pode ser algo problemático. Neste e no próximo slide apresentam-se algumas soluções.

```c
#include <stdio.h>

int main(void) {
   int n1=-1, n2=-1, n3=-1;
   int *ptr=&n3;

   printf("Enter an integer number: ");
   scanf("%d", &n1); /* scanf simples para a introdução de um número */

   printf("Enter an integer number: ");
   scanf("%d%*c", &n2); /* melhor, pois remove o ENTER de forma a limpar o input buffer */

   printf("Enter another integer number: ");
   scanf("%*[^0-9]%d%*[^\n]%*c", ptr); /* rejeita tudo exceto o primeiro número */

   printf("You entered: %d, %d and %d\n", n1, n2, n3);

   return 0;
}
```

Sequência de controlo do **scanf**:
- **%d**: recebe um inteiro
- **%*c**: rejeita um caracter
- **%*[^0-9]**: rejeita tudo exceto números
- **%*[^\n]**: rejeita tudo exceto o '\n'

Indica-se ao **scanf** o endereço de memória onde o *input* deverá ser armazenado

Console I/O: Input

- O **`scanf`** consegue ser um bocado problemático…
- O código seguinte apresenta uma outra opção
  - Ler uma *string* e convertê-la conforme necessário

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

int main(void) {
   int n=-1;
   char c, buf[BUFSIZ]; /* BUFSIZ está definido em stdio.h */

   while( (c=getchar())!='\n' && c!=EOF); /* ciclo para limpeza do stdin buffer */

   printf("Enter an integer number: ");

   if(fgets(buf, sizeof(buf), stdin) != NULL) {
      n = atoi(buf); /* devolve 0 se a conversão falhar */
      printf("You entered: %d\n", n);
      return EXIT_SUCCESS;
   }

   return EXIT_FAILURE; /* se chegar aqui, o fgets falhou */
}
```

Neste exemplo não é necessário limpar o *input buffer*, pois não tinha sido usado previamente.
Se tivessemos utilizado anteriormente o **`scanf`**, sem os devidos cuidados, seria boa ideia limpar o *input buffer*.

Desta forma não se consegue distinguir a inserção do número zero, de qualquer outra inserção que gere erro.

Para distinguir entre esses casos, será necessário testar a *string* `buf` antes de invocar a função `atoi()`. Para uma proposta de solução, ver próximo slide…

Console I/O: Input

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

/* devolve 1 caso a string seja um número, 0 caso contrário */
int is_number(char *str) {
   if(str==NULL) return 0;

   int i=0;
   if(*str=='-') ++i;
   while( *(str+i)!='\n' && *(str+i)!='\0' ) {
      if( '0'<=*(str+i) && *(str+i)<='9' ) ++i;/* código ASCII do char corresponde a um dígito */
      else return 0;
   }
   return 1;
}


int main(void) {
   int n=-1;
   char c, buf[BUFSIZ]; /* BUFSIZ está definido em stdio.h */

   printf("Enter an integer number: ");

   if(fgets(buf, sizeof(buf), stdin)!=NULL && is_number(buf)) {
      n = atoi(buf);
      printf("You entered: %d\n", n);
      return EXIT_SUCCESS;
   }

   return EXIT_FAILURE; /* se chegar aqui não se conseguiu obter um número do utilizador */
}
```

Exercícios

1. Escreva a representação na memória, em Big Endian e Little Endian dos seguintes valores:
   a) `0x1188` (16 bits)
   b) `0xFF3455B6` (32 bits)
   c) `0x28934DEF` (32 bits)

2. Corrija e melhore a qualidade do código:

```
int main(void) {
    int * ptr  ;
    int i;
    int soma=0;

    for(i=0; i<10; ++i)
        {
            scanf("%d", ptr);
            soma = soma + *ptr;
        }

    printf("Soma = %d \n", soma);

    return 0;
}
```

Bibliografia

- Instituto Superior de Engenharia do Porto. Computer Architecture. Luís Nogueira. “Pointers in C”. 2024/2025. 19 Diapositivos. https://moodle.isep.ipp.pt/mod/resource/view.php?id=216577