UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Ponteiros & Endereços

Aula 13 | versão 1.2

Marcos Silvano Almeida
marcossilvano@professores.utfpr.edu.br
Departamento de Computação
UTFPR Campo Mourão

Roteiro

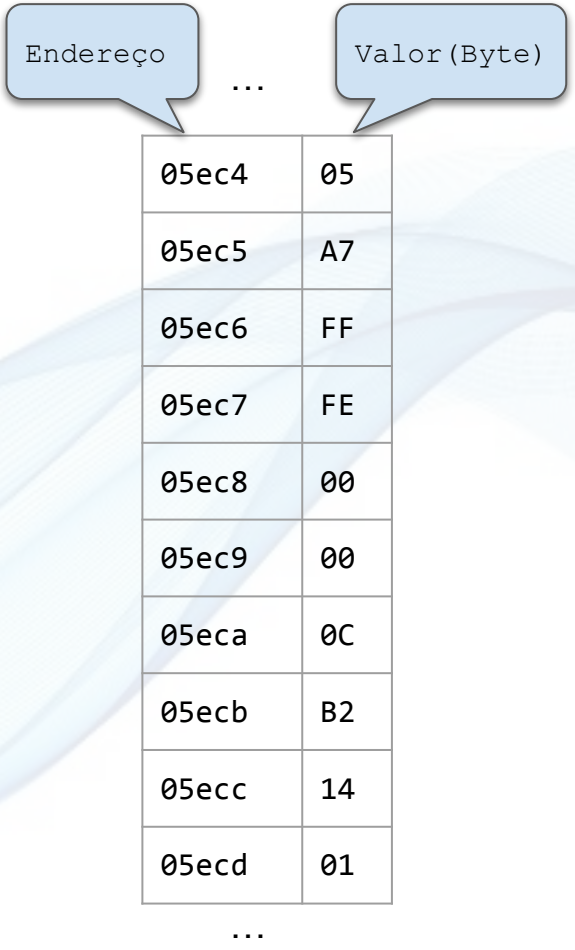
- Layout da memória
- Endereços de variáveis
- Ponteiros para endereços
- Aritmética de ponteiros
- Ponteiros para funções

Layout da memória

- No computador, a memória primária (RAM/ROM) é acessada por endereços

- Cada endereço refere-se a um Byte.
  - Por essa razão, diz-se que a memória no computador é "**byte-endereçável**".

- Por conveniência, o exemplo ao lado está mostrando os **endereços** e **valores** dos bytes em hexadecimal
  - `Hex: 0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, F`
  - `Dec: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15`
    - `1 dígito hex  = 4 bits (nibble)`
    - `2 dígitos hex = 8 bits (byte)`



Layout da memória

- As variáveis são abstrações de células de memória
  - Abstrações = representações simplificadas
  - As variáveis estão armazenadas em posições (células) de memória

- Cada tipo de dados especifica o tamanho em bytes
  - Podemos verificar o tamanho com operador **sizeof()**

```
Ex: sizeof(int), sizeof(soma), sizeof(unsigned short int)
```

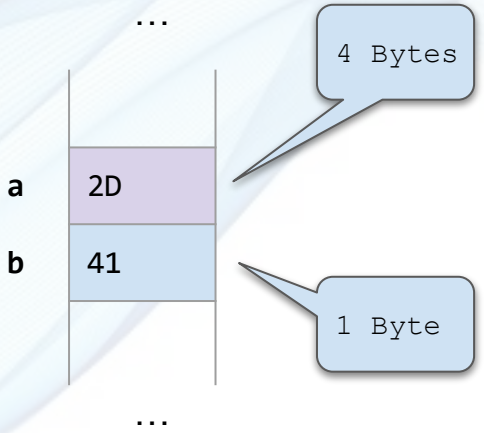
char	[–128,127]	1 byte/8 bits
short int	[–32,767, +32,766]	2 bytes/16 bits
int	[–2,147,483,647, +2,147,483,646]	4 bytes/32 bits
long	[–9,223,372,036,854,775,807, +9,223,372,036,854,775,806]	8 bytes/64 bits

Layout da memória

- Célula **abstrata**
  - Consideramos uma célula para cada variável
  - Nesta representação, cada célula tem o tamanho da variável

```
int a = 45;    // 45 = $2D (4 bytes)

char b = 'A'; // 65 = $41 (1 byte)
```

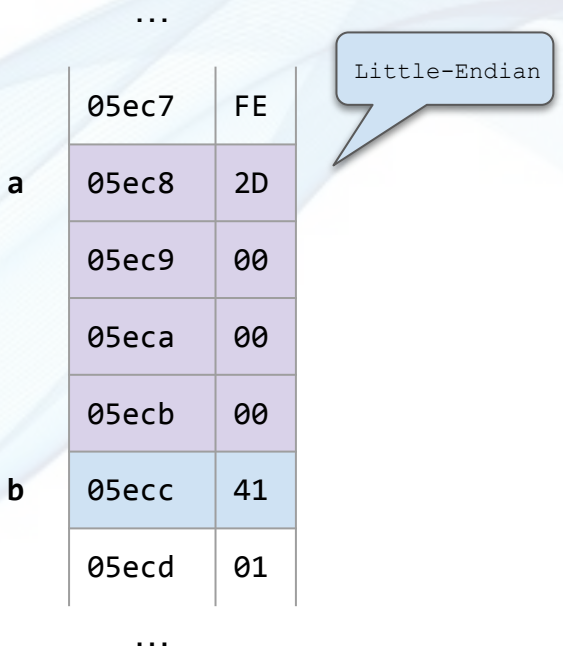


Layout da memória

- Célula **física**
  - Consideramos o uso real de memória: cada célula tem o tamanho de 1 byte
  - Cada variável ocupa uma ou mais bytes, isto é, uma ou mais posições de memória

```
int a = 45;    // 45 = $2D (4 bytes)

char b = 'A'; // 65 = $41 (1 byte)
```



Endereços de Variáveis Tipo Valor

&

Endereços de variáveis

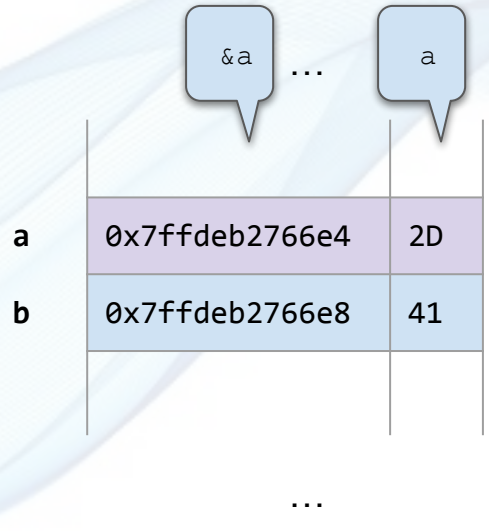
- Para acessarmos o endereço (célula física) de uma variável **Tipo Valor**, utilizamos o operador **&**:

  &variável

```
int a = 45;
char b = 'A';

printf(" a: %08X\n", a);   //$0000002D
printf("&a: %p\n", &a);    //0x7ffdeb2766e4

printf(" b: $%02X\n", b); //$41
printf("&b: %p\n", &b);    //0x7ffdeb2766e8
```



Variáveis Tipo Endereço: Ponteiros

*

Ponteiros

- Uma variável ponteiro armazena um endereço de memória
  - Ponteiro ⇒ **Tipo Endereço**

- Sintaxe:

```
int  *p_int;     << ponteiro para int (acessa 4 bytes)
char *p_char;    << ponteiro para char (acessa 1 byte)
long *p_long;    << ponteiro para long (acessa 8 bytes)
```

O tipo indica o tamanho do dado que deve ser acessado.

```
int num = 5;

int *p_num = &num;
```

	Endereço	Valor
num	0x7ffde	5
p_num	0x7ffe2	0x7ffde

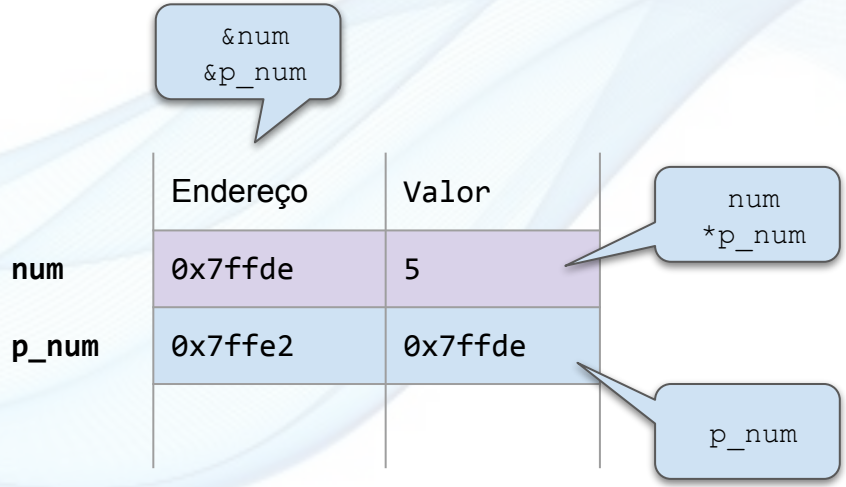
Ponteiros

- Imprimindo valores e endereços
  - Acesso ao endereço de uma variável: operador &
  - Acesso ao endereço contido em um ponteiro: operador * (desreferenciamento)

```
int num = 5;
int *p_num = &num;

printf(" num: %d\n", num);
printf(" &num: %p\n\n", &num);

printf(" *num: %d\n", *p_num);
printf("p_num: %p\n", p_num);
```



Resumo

- Obter endereço de uma variável……..…: **&var**

```
int a = 5;
printf(" endereço de 'a': %p\n\n", &a);
```

- Armazenar o endereço de 'a' em uma variável ponteiro..: **pointer = &var**

```
int *p_a = &a;
printf(" conteúdo de 'p_a': %p\n\n", &p_a);
```

- Acessar o conteúdo da variável por meio do ponteiro…..: ***pointer**

```
printf(" conteúdo de 'a': %d\n", *p_a);
```

Funções:

Passagem de parâmetros por **valor** vs **endereço**

Passagem de parâmetro por valor

⇒ **Por valor**: parâmetro de entrada

- A função recebe uma cópia do valor passado aos parâmetros.

```c
// Verifica se número é par (true/false)
int is_even(int num) {
    return num % 2 == 0;
}

int main() {
    printf("Even ? %d\n", is_even(568));
    printf("Even ? %d\n", is_even(1021));
    return 0;
}
```

Passagem de parâmetro por endereço

⇐⇒ **Por endereço**: parâmetro de entrada e saída

- A função recebe um parâmetro com o endereço da variável passada à função

```
// Recebe três números e retorna, via parâmetros, o maior e o menor
void min_max(int a, int b, int c, int* min, int* max) {
    *min = a, *max = a;
    if (b < *min) *min = b;
    if (c < *min) *min = c;
    if (b > *max) *max = b;
    if (c > *max) *max = c;
}
int main() {
    int min, max;
    min_max(201, 156, -23, &min, &max);
    printf("Menor: %d, Maior: %d \n", n, m);
    return 0;
}
```

Passagem de parâmetro por endereço

⇐⇒ **Por endereço**: parâmetro de entrada e saída

- Ao utilizar ponteiros como parâmetros em funções, podemos modificar os valores de variáveis passadas como argumentos, na chamada da função

```
// Inverte os valores das variáveis
void swap(int *p1, int *p2) {
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
int main() {
    int a = 5;
    int b = 17;
    printf("a: %d, b: %d\n", a, b); // a:5, b:17
    swap(&a, &b);
    printf("a: %d, b: %d\n", a, b); // a:17, b:5
}
```

Aritmética de ponteiros

Operadores de ponteiros

- Podemos utilizar operadores aritméticos para calcular deslocamentos
- Operadores comuns:
  - \+ incrementa o endereço em uma posição (char, int, long, etc)
  - – decrementa o endereço em uma posição (char, int, long, etc)

```
int v1[] = {2,4,8,16,32,64};

int *p = v1;

printf("%d ", *p); // 2
p += 1; // incrementa posição para o próximo inteiro

printf("%d ", *p); // 4
p += 3; // incrementa posição para três inteiros à frente
printf("%d\n", *p); // 32
```

Como o ponteiro é para inteiros, + e – deslocam o endereço em inteiros (4 bytes)

Operadores de ponteiros

- Podemos utilizar operadores aritméticos para calcular deslocamentos
- Variáveis arrays são ponteiros

```
int v1[] = {2,4,6,8};

//arrays/ponteiros aceitam estas duas formas de acesso
printf("%d ", v1[0]);
printf("%d ", v1[1]);
printf("%d ", v1[2]);
printf("%d\n", v1[3]);

printf("%d ", *(v1+0));
printf("%d ", *(v1+1));
printf("%d ", *(v1+1));
printf("%d\n",*(v1+3));
```

Acessa o endereço em V1
+ deslocamento de X inteiros

UTFPR

Percorrendo uma string

- É comum utilizarmos ponteiros para percorrer vetores, especialmente strings.

```
char str[] = "Ponteiros em C";

char *p = str; // p "aponta" para o primeiro char de str

while (*p != 0) {
    printf("%c ", *p);
    p++;
}

printf("\n");
```

"Hackeando" a matriz

- Vetores bidimensionais alocados localmente são, essencialmente, vetores de uma dimensão com as "linhas" em sequência.

```
int v[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9,10,11,12}
};

int *p = (int*)v;
for (int i = 0; i < 12; i++) {
    printf("%d ", *p);
    p++;
}
printf("\n");
```

Exemplo: função que imprime vetor de forma reversa

- Podemos utilizar ponteiros para percorrer vetores/arrays, empregando operadores de ponteiros para realizar deslocamentos dos mesmos sobre os endereços dos elementos no vetor

```
// n: número de elementos, *v: endereço do primeiro elemento no vetor
void print_rev(int n, int* v) {
    v += n-1;      // posiciona v no endereço do último elemento
    for (int i = 0; i < n; i++) {
        printf("%d ", *v);
        v--;
    }
    printf("\n");
}

int main() {
    print_reverse(10, (int[]){2,4,6,8,10,12,14,16,18,20});
    ...
```

Ponteiros para Structs

Ponteiros para structs

- Para acessar os campos de um struct via ponteiro, utiliza-se "->"

```
struct object {
    char name[51];
    float width;
    float height;
};
typedef struct object Object;

void test() {
    Object obj = {"square", 45, 45};

    Object* p_obj = &obj; // p_obj aponta para obj (recebe o endereço de obj)

    // alterando os campos de obj via ponteiro p_obj
    p_obj->width = 50;
    p_obj->height= 50;
    strcpy(p_obj->name, "box");
}
```

Ponteiros para structs

- Para acessar os campos de um struct via ponteiro, utiliza-se "->"

```
void print(const Object* obj) {
    printf("{%s, %d, %d}\n", obj->name, obj->width, obj->height);
}

void reset(Object* obj) {
    obj->width = 0;
    obj->height= 0;
    strcpy(obj->name, "");
}

void test() {
    Object obj = {"rect", 95, 45};
    Object* p_obj = &obj; // p_obj aponta para obj (recebe o endereço de obj)

    print(p_obj);   // {"rect", 95, 45}
    reset(&obj);
    print(p_obj);   // {"none", 0, 0}
}
```

'const' avisa que a função não fará alterações no struct de endereço informado

Ponteiros de Funções

Ponteiro para função

- Podemos definir ponteiro para função, isto é, uma variável que armazena o endereço de uma função e pode ser usada para chamá-la.
- Sintaxe:

```
tipo_retorno (*var_ponteiro) (params…)
```

- Exemplo:

```
void print_hello() {
    printf("Hello!\n");
}

int main() {
    void (*p_func)() = print_hello;
    p_func();
    return 0;
}
```

Ponteiro para função: params e retorno

- Exemplo de ponteiro para função que possui parâmetros e retorna valor

```
int max(int a, int b) {
     if (a > b)
          return a;
     else
          return b;
}

int main() {
     int (*p_max)(int,int) = max;
     printf("max: %d\n", p_max(5,9));
     return 0;
}
```

Ponteiro de função: função como parâmetro

- Um exemplo interessante é criar uma função que varre um vetor e chama uma determinada função para cada elemento do vetor

```c
void for_each(int n, int* v, int (*func)(int)) {
    for (int i = 0; i < n; i++) {
        v[i] = func(v[i]);
    }
}

int main() {
    int v[] = { 1,-2, 3, 4,-5, 6, 7,-8, 9,10};
    // altera neg >> pos e imprime o vetor
    for_each(10, v, make_positive);
    for_each(10, v, print_elem);
    return 0;
}
```

```c
int print_elem(int a) {
    printf("%d ", a);
    return a;
}

int make_positive(int a) {
    return a > 0? a : -a;
}
```

Exemplo: parametrizando critério de ordenação

- Outro exemplo interessante é criar uma função que ordena um vetor e definir o critério da ordenação em outra função parametrizável

```
int asc (int a, int b)     {return a < b;}
int desc (int a, int b)    {return a > b;}
int shuffle(int a, int b) {return rand() % 2;}

void sort(int n, int* v, int (*comp)(int, int)) {
   for (int i = 0; i < n; i++) {
       for (int j = i+1; j < n; j++) {
           if (comp(v[j], v[i])) {
               int temp = v[i];
               v[i] = v[j];
               v[j] = temp;
           }
       }
   }
}
```

```
int main() {
   int v[] = {6,3,4,3,2,1};

   sort1(6, v, asc);
   print_vector(6, v);

   sort1(6, v, desc);
   print_vector(6, v);

   sort1(6, v, shuffle);
   print_vector(6, v);
   return 0;
}
```

Exemplo: parametrizando critério de ordenação em structs

- Definimos qual **campo** e **ordem** utilizar para ordenar um vetor de structs

```
struct entity {
   int id;
   char name[51];
   int age;
};
typedef struct entity Entity;

int main() {
   Entity v[] = { {5,"Five",5}, {7,"Seven",7}, {3,"Three",3}, {6,"Six",6}
                  {1,"One",1},  {2,"Two",2},   {4,"Four",4},  {8,"Eight",8} };
   sort(8, v, asc_by_id);
   print(8, v);
   sort(8, v, asc_by_name);
   print(8, v);
   return 0;
}
```

Exemplo: parametrizando critério de ordenação em structs

```
int asc_by_id(Entity a, Entity b) {
    return a.id < b.id;
}
int asc_by_name(Entity a, Entity b) {
    return strcmp(a.name, b.name) < 0;
}
void sort(int n, Entity* v, int (*comp)(Entity, Entity)) {
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            if (comp(v[j], v[i])) {
                Entity temp = v[j];
                v[j] = v[i];
                v[i] = temp;
            }
        }
    }
}
```

Referências

- Algoritmos e Programação
  - Marcela Gonçalves dos Santos
  - Disponível pelo Moodle

- Estruturas de Dados, Waldemar Celes e José Lucas Rangel
  - PUC-RIO - Curso de Engenharia
  - Disponível pelo Moodle

- Linguagem C, Silvio do Lago Pereira
  - USP - Instituto de Matemática e Estatística
  - Disponível pelo Moodle

- Curso Interativo da Linguagem C
  - https://www.tutorialspoint.com/cprogramming