Ponteiros

Prof. Marcelo Veiga Neves
marcelo.neves@pucrs.br

Introdução

- Cada variável em um programa C é armazenada em algum lugar na memória
- A linguagem possui um operador para obter esses endereços: o operador &
- Por exemplo, vamos exibir o endereço de uma variável:

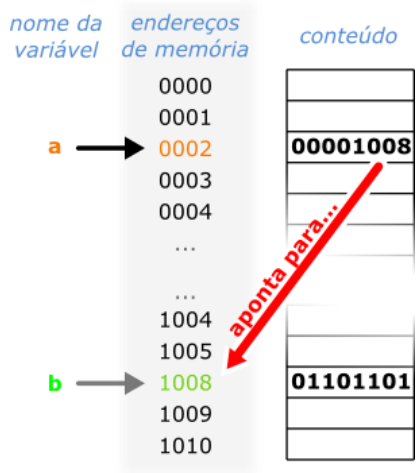
```c
int main()
{
    int b = 200;
    printf("%p\n", &b);
    printf("%llx\n", (unsigned long long)&b);

    return 0;
}
```

- O operador & retorna o endereço do seu operando
- O modificador %p exibe o valor como um endereço de memória em hexa
- No segunda caso, explicitamente o endereço está sendo convertido para um inteiro e impresso em hexa

Ponteiro

- Definição: uma variável que armazena um endereço de memória (usualmente de outra variável)



- Por exemplo: criando um ponteiro para a variável *b* (chamado *a*):

```
int main()
{
    int b = 200;
    int *a = &b;
    ...
}
```

- O operador * declara que *a* é um *ponteiro para um int*

Entendendo os operadores de ponteiro

- O operador & retorna o endereço de algo
- O operador * é usado para declarar um ponteiro ou acessar os conteúdos de um endereço de memória:

```
int main()
{
    int b = 200;
    int *a = &b;     // a armazenará o endereço de b

    printf("Valor de b: %d\n", b);
    printf("Valor de a: %p\n", a);
    printf("Conteúdo apontado por a: %d", *a);
    ...
}
```

- Quando usado **à esquerda** de uma variável, o operador * é chamado de *operador de dereferência*

Ponteiros podem ser maus!

- Precisamos ter certeza de sempre usar ponteiros válidos
- Um ponteiro válido é um ponteiro que aponta para uma área de memória válida (usualmente uma variável no programa)
- Se esquecermos de atribuir um valor ao ponteiro e tentarmos usá-lo, coisas ruins podem acontecer!

```
int main()
{
    int a = 200;
    int *b;

    printf("Conteúdo apontado por b: %d\n", *b);    //
    oops!
    ...
}
```

- Este código provavelmente irá gerar uma falha de segmentação no Linux

O ponteiro NULL

- É considerada uma prática saudável armazenar o valor NULL em ponteiros não inicializados
- NULL é simplesmente uma constante definida como ZERO
- No código, pode ser usado com segurança para testar se um ponteiro está inicializado ou não:

```
int main()
{
    int a = 200;
    int *b = NULL;

    ...
    if (b)
        printf("Conteúdo apontado por b: %d\n", *b);    //
    b é válido
    else
        printf("Ponteiro inválido!\n");
    ...
}
```

Ponteiros e *arrays*

- Em C, um *array* é armazenado como um ponteiro para o seu primeiro elemento
- Os demais elementos são armazenados contiguamente a partir do endereço inicial
- Por exemplo:

```
char c[10];
```

- *c* é um ponteiro para uma área de memória capaz de armazenar 10 *char*
- Portanto, o tamanho da área de memória é *10 * sizeof(char) = 10 bytes*

Arrays são sempre passados por referência

- Uma consequência interessante dessa implementação é que os arrays são sempre passados por referência:

```
void incArray(int v[])
{
    int c;
    for (c = 0; c < 10; c++)
        v[c]++;
}

int main()
{
    int data[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    incArray(data);
}
```

- Ou podemos escrever a função desta forma:

```
void incArray(int *v)
```

Aritmética de ponteiros

- Considerando o seguinte *array de char*:

```
char c[10] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
'i', 'j'};
```

- Podemos declarar um ponteiro p para ele:

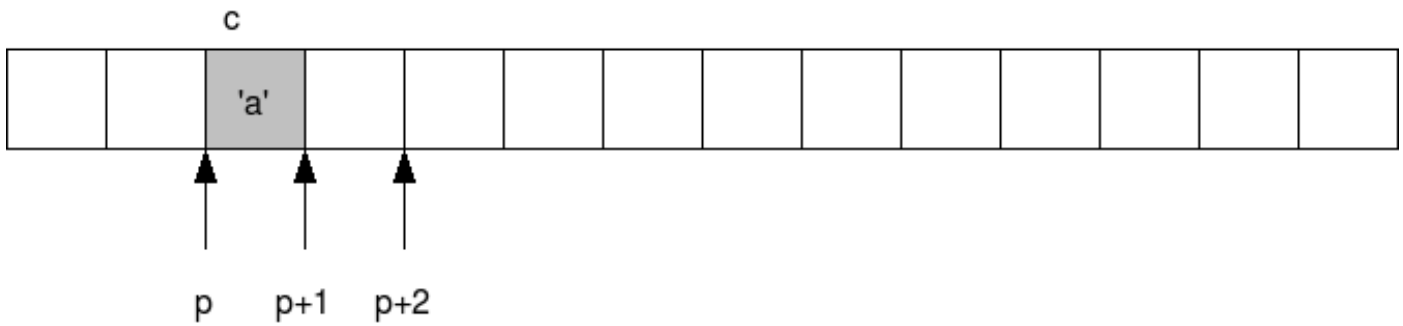
```
char *p = c;    // não precisa &, c já é um ponteiro
```

E agora *p* é o mesmo que *c*:

```
printf("%c\n", *p); // exibe o primeiro elemento do
array
printf("%c\n", p[0]);   // idem
...
printf("%p\n", p);  // exibe o endereço do primeiro
elemento
printf("%p\n", c);  // idem
printf("%p\n", &c[0]);  // idem
```

Aritmética de ponteiros

- A aritmética de ponteiros nos permite acessar outros elementos através do operador de adição:

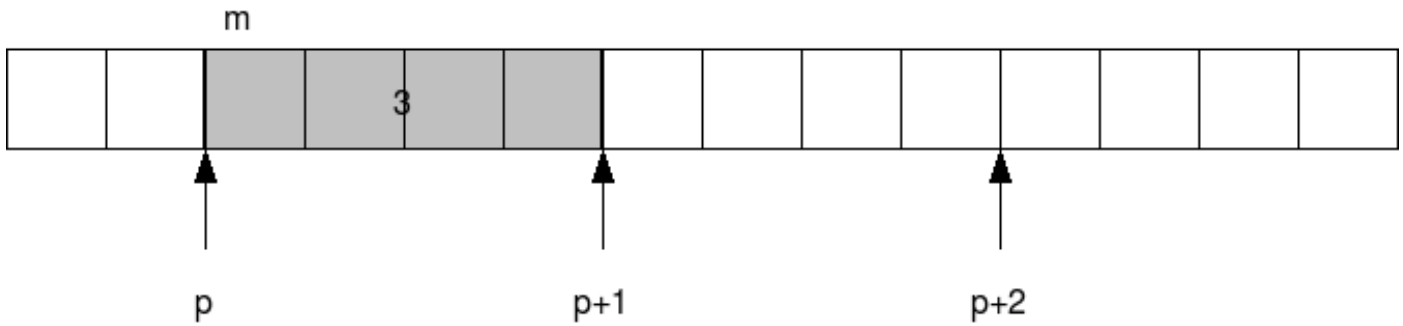



```
...
char *p = c;

printf("%c\n", *p);      // exibe 'a'
printf("%c\n", *(p+1)); // exibe 'b'
printf("%c\n", *(p+2)); // exibe 'c'
printf("%c\n", *(p+3)); // exibe 'd'
...
```

Aritmética de ponteiros é inteligente!

- Se o array armazena *ints*, o próximo elemento estará a 4 bytes de distância[1]
- O tamanho da área de memória será *10 * sizeof(int) = 40 bytes*




```
int m[] = {3, 2, 1, 4, 5, 6, 9, 8, 7, 10};
printf("%p\n", m);      // end. do primeiro elem., e.g.
0x7fff5fbfebf8
printf("%p\n", &m[0]); // idem
printf("%p\n", m+1);    // end. do segundo elem., e.g.
0x7fff5fbfebfc
printf("%p\n", &m[1]); // idem
...
```

[1] Considerando que um *int* tenha um tamanho de 32 bits, o que pode não ser o caso em algumas arquiteturas.

Experimento: aritmética de ponteiros

- Compare os seguintes programas:

```c
#include <stdio.h>

int main()
{
    int vet[] = {4, 9, 12};
    int i, *ptr;

    ptr = vet;
    for (i = 0; i < 3; i++)
        printf("%d ",
    *ptr++);

    return 1;
}
```

```c
#include <stdio.h>

int main()
{
    int vet[] = {4, 9, 12};
    int i, *ptr;

    ptr = vet;
    for (i = 0; i < 3; i++)
        printf("%d ",
    (*ptr)++);

    return 1;
}
```

- Você pode explicar o que é exibido na tela em cada caso?

Exercícios

1. Considere o seguinte código, que exibe o conteúdo do array junto ao endereço de memória de cada elemento:

```
int main()
{
    int nums[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int *ptr = nums;
    int i, bytes;

    for (i = 0, bytes = 0; i < 10; i++, bytes += 4)
        printf("%d: %p + %d bytes = %p ==> %d\n",
        i, ptr, bytes, (ptr+i), *(ptr+i));

    return 0;
}
```

   Você consegue alterar o código, fazendo que ele exiba o conteúdo de cada byte do array?

Exercícios

2. Você consegue exibir o conteúdo das seguintes varíaveis (bytes armazenados na memória), usando apenas ponteiros para *char* (bytes)?

```
int main() {
    char *text = "Prog. Software Basico";
    int v[] = {0, 1, 2, 3, 4};
    int matriz[4][4] = {
        {0, 1, 2, 3},
        {4, 5, 6, 7},
        {8, 9, 10, 11},
        {12, 13, 14, 15}
    };
}
```

Referências

- Este material foi construído pelo Prof. Sérgio Johann Filho com base nos slides de aula do prof. Marcelo Cohen