ACH2001 - Introdução à Programação

Ponteiros (e o que eles tem a ver com vetores)

Prof. Flávio Luiz Coutinho
flcoutinho@usp.br

Ponteiros

Variáveis de tipo “ponteiro” guardam um endereço de memória.

Ponteiros

Variáveis de tipo “ponteiro” guardam um endereço de memória.

Suponha que eu queira guardar o endereço e uma variável inteira **x**. Como fazer?

Ponteiros

Variáveis de tipo “ponteiro” guardam um endereço de memória.

Suponha que eu queira guardar o endereço e uma variável inteira **x**. Como fazer?

```
int x = 255;
int * p = &x;          // variável p
                       // tipo int * (endereço de um valor int)
                       // p recebe o endereço de x
```

Ponteiros

O que eu posso fazer a partir do endereço de memória na variável **p**?

Ponteiros

O que eu posso fazer a partir do endereço de memória na variável **p**?

Acessar o valor guardado no endereço de memória indicado pelo ponteiro.

Ponteiros

O que eu posso fazer a partir do endereço de memória na variável **p**?

Acessar o valor guardado no endereço de memória indicado pelo ponteiro.

```
int y = *p;   // valor que consta no endereço indicado em p
              // é atribuído na variável y.

printf("valor = %d", y)    // qual o valor impresso na saída?
```

Ponteiros

Dois operadores:

&x: obtém o endereço da variável **x**

*p: valor armazenado no endereço apontado por **p**

Ponteiros

Quando isso pode ser útil?

Ponteiros

Quando isso pode ser útil? Quando queremos “mudar o valor” de um parâmetro passado para uma função:

Ponteiros

Quando isso pode ser útil? Quando queremos “mudar o valor” de um parâmetro passado para uma função:

```
void duplica(int * p){
    *p = 2 * (*p);
}
```

```
int main(){
        int x = 255;
        duplica(&x);
        printf(“x = %\n”, x);
        return 0;
    }
```

Ponteiros

Quando isso pode ser útil? Quando queremos “mudar o valor” de um parâmetro passado para uma função:

```
void duplica(int * p){
    *p = 2 * (*p);
}
```

```
int main(){
        int x = 255;
        duplica(&x);
        printf(“x = %\n”, x);
        return 0;
    }
```

Mas atenção: isso deve ser evitado sempre que possível...

Ponteiros

A boa prática é sempre escrever funções que não alteram os valores dos parâmetros passados na chamada (ou seja, não tenham efeitos colaterais):

```
int duplica(int n){
    return 2 * n;
}
```

```
int main(){
    int x = 255;
    int y = duplica(x);
    printf("x = %\n", y);
    return 0;
}
```

Ponteiros

A boa prática é sempre escrever funções que não alteram os valores dos parâmetros passados na chamada (ou seja, não tenham efeitos colaterais):

```
int duplica(int n){
    return 2 * n;
}
```

```
int main(){
    int x = 255;
    int y = duplica(x);
    printf("x = %\n", y);
    return 0;
}
```

Mas às vezes isso pode necessário. Exemplo: função scanf.

Ponteiros

E o que ponteiros tem a ver com vetores (*arrays*)?

Ponteiros

E o que ponteiros tem a ver com vetores (*arrays*)?

Tudo a ver!

Ponteiros

E o que ponteiros tem a ver com vetores (*arrays*)?

Tudo a ver!

Um variável do tipo vetor, é um ponteiro!

Ponteiros

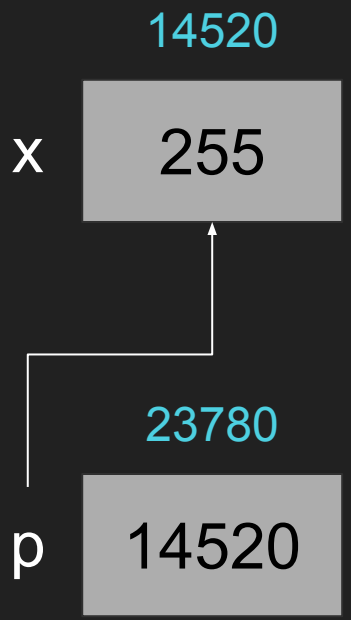
E o que ponteiros tem a ver com vetores (*arrays*)?

Tudo a ver!

Um variável do tipo vetor, é um ponteiro!

Uma variável do tipo vetor guarda o endereço do início do bloco de memória reservado para este.

Ponteiros




```
int x = 255;

int * p = &x;
```

Ponteiros

```
int a[5] = { 1, 2, 3, 4, 5 };
printf("%d\n", a[3]);
printf("%d\n", *(a + 3));
```

Ponteiros

```
int i;

int * p;

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


for(i = 0; i < 5; i++) printf("%d\n", a[i]);

for(i = 0; i < 5; i++) printf("%d\n", *(a + i));

for(p = a; p < a + 5; p++) printf("%d\n", *p);
```

Alocação dinâmica de memória

Alocação estática de vetores possui duas desvantagens:

- aumento da capacidade exige mudança e recompilação do programa.
- a memória consumida pelo programa é sempre a mesma, independente da quantidade que é efetivamente usada.

Alocação dinâmica de memória

Alocação estática de vetores possui duas desvantagens:

- aumento da capacidade exige mudança e recompilação do programa.
- a memória consumida pelo programa é sempre a mesma, independente da quantidade que é efetivamente usada.

- vetor estático de tamanho pequeno: capacidade limitada de guardar dados
- vetor estático de tamanho grande: provável desperdício de memória.

Alocação dinâmica de memória

Alocação estática de vetores possui duas desvantagens:

- aumento da capacidade exige mudança e recompilação do programa.
- a memória consumida pelo programa é sempre a mesma, independente da quantidade que é efetivamente usada.

- vetor estático de tamanho pequeno: capacidade limitada de guardar dados
- vetor estático de tamanho grande: provável desperdício de memória.

O ideal seria usar apenas o necessário para uma certa execução do programa.

Alocação dinâmica de memória

Com fazer isso? Com alocação dinâmica de vetores.

Função **malloc**: aloca um bloco de memória com **n** bytes, e devolve o endereço do início deste bloco de memória alocado.

```
int * a = (int *) malloc (tamanho * sizeof(int));
```

Alocação dinâmica de memória

```
int * a = (int *) malloc (10 * sizeof(int));
```

Liberação de memória

Diferentemente de variáveis locais de funções, que são “liberadas” quando uma chamada da função termina, blocos de memória alocados de forma dinâmica devem ser “liberados” quando não precisarem mais ser utilizados.

Liberação de memória

Diferentemente de variáveis locais de funções, que são “liberadas” quando uma chamada da função termina, blocos de memória alocados de forma dinâmica devem ser “liberados” quando não precisarem mais ser utilizados.

```
int * a = (int *) malloc (tamanho * sizeof(int));

// uso do vetor alocado de forma dinâmica

free(a);
```

Liberação de memória

Ao final da execução de um programa, toda a memória que tiver sido reservada a este será liberada de qualquer forma.

Liberação de memória

Ao final da execução de um programa, toda a memória que tiver sido reservada a este será liberada de qualquer forma.

Entretanto se um programa aloca diversos blocos de memória, e não libera aqueles que não terão mais uso, haverá um consumo excessivo e desnecessário de memória, que pode esgotar os recursos da máquina.

Liberação de memória

Ao final da execução de um programa, toda a memória que tiver sido reservada a este será liberada de qualquer forma.

Entretanto se um programa aloca diversos blocos de memória, e não libera aqueles que não terão mais uso, haverá um consumo excessivo e desnecessário de memória, que pode esgotar os recursos da máquina.

Isso é conhecido como vazamento de memória (*memory leak*).