

Ponteiros

Um ponteiro é uma variável que armazena o endereço de outra variável. Ponteiros devem ser declarados com o símbolo `*`:

```
int *x; /* ponteiro para int */
char *s; /* ponteiro para char */
int **y; /* ponteiro para ponteiro para int */
```

O operador `&` obtém o endereço de uma variável, enquanto o operador `*` acessa (dereferencia) o que está no endereço de memória cujo valor é aquela expressão. Por exemplo:

```
int x = 5;
int *y;
y = &x; /* o valor de y passa a ser o endereço de x */
*y = 3; /* escreve 3 no endereço cujo valor é y */
printf("%d\n", x);
```

Imprime 3. Mesmo sem alterar `x` diretamente, alteramos o que está na memória em que `x` está armazenado, o que tem o mesmo efeito.

Note que o `*` tem significados diferentes quando usado na declaração (`int *y`) e quando usado como operador (`*y = 3;`): no primeiro caso, o asterisco serve apenas para denotar que a variável `y` é um ponteiro. Note também que os operadores `&` e `*` são como funções inversas, de modo que `&*x` equivale a `*&x` que equivale a `x`.

Como outras variáveis, é possível atribuir um valor a um ponteiro durante sua declaração; por exemplo, as duas linhas do exemplo acima

```
int *y;
y = &x;
```

Podem ser escritas como uma só:

```
int *y = &x;
```

Erros ao manipular ponteiros são comuns, e são a causa mais comum do famoso Segmentation fault.

Nota: em uma declaração como

```
int *x, y;
```

`x` é um ponteiro para `int`, mas `y` é apenas um `int`, não um ponteiro. Isso talvez não seja surpreendente quando a declaração é escrita assim, mas essa declaração poderia também ser escrita como

```
int* x, y;
```

Ainda assim, `y` continua não sendo um ponteiro.

NULL

`NULL` é um valor especial que indica que o ponteiro não aponta para nenhum endereço válido. É uma boa prática atribuir um ponteiro a `NULL` durante sua declaração. Dereferenciar um ponteiro que aponta para `NULL` gera um Segmentation fault:

```
int *p = NULL;
int x = *p + 1; /* Segmentation fault! */
```

O valor `NULL` é quase sempre definido como `0`, de modo que a condição

```
if(p == NULL) {
    /* ... */
}
```

É frequentemente escrita como

```
if(!p) {
    /* ... */
}
```

E da mesma forma, `if(p != NULL)` pode ser escrito simplesmente como `if(p)`.

Nota: `NULL` é garantidamente `0` em sistemas que seguem a especificação POSIX, entre os quais Linux.

Relação com arrays

Arrays e ponteiros estão fortemente relacionados. Uma array, na verdade, guarda o endereço de seu primeiro elemento:

```
int arr[20];
int *p;
p = arr; /* válido: p agora aponta para o primeiro elemento de arr */
```

Além disso, é possível somar um valor inteiro a um ponteiro para obter o n-ésimo valor depois daquele endereço:

```
int arr[20];
int *p;

p = arr + 4; /* o endereço 4 ints depois do começo de arr; ou seja, o endereço do quinto elemento de arr */
```

Além disso, por definição, `arr[i]` é equivalente a `*(arr + i)`. Desse modo, se declaramos uma array como

```
double arr[SIZE];
```

Podemos percorrer seus elementos como

```
int i;
for(i = 0; i < SIZE; i++) {
    /* fazer algo com arr[i] */
}
```

Ou como

```
double arr[SIZE];
/* ... */
double *p;
for(p = arr; p < arr + SIZE; p++) {
    /* fazer algo com *p */
}
```

Isso explica porque os índices das arrays em C começam em zero: `arr[0]` equivale a `*(arr + 0)`, que equivale a `*arr`. Isso explica também porque índices negativos não funcionam em C da mesma forma que em Python: `arr[-4]` é o endereço do quarto elemento **antes** do começo da array. Por último, isso explica porque `&arr[2]` é equivalente a `arr + 2`, pois `&arr[2] -> &*(arr + 2) -> arr + 2`.

Diferenças com arrays

Arrays e ponteiros são semelhantes, mas não idênticos. Primeiramente, não é possível mudar o endereço que a array guarda:

```
int arr[20];  
  
int x;  
int *p;  
  
p = &x; /* OK */  
arr = &x; /* ERRO! */  
  
p++; /* OK */  
arr++; /* ERRO! */
```

Mas a diferença mais importante aparece ao usar o operador `sizeof`:

```
int arr[20];  
int *p = arr;  
  
printf("%d %d\n", sizeof(arr), sizeof(p));
```

Imprime valores diferentes: `sizeof(arr)` é igual a `20 * sizeof(int)`, enquanto `sizeof(p)` corresponde a `sizeof(int*)`.

Alocação dinâmica

Arrays possuem uma limitação séria: seu tamanho deve ser estático e conhecido em tempo de compilação. Por exemplo, o código a seguir **não** é válido:

```
size_t size;  
/* leia size de algum lugar */  
int p[size]; /* ERRO! */
```

Para contornar essa limitação, usa-se a alocação dinâmica de memória, por meio das funções

`malloc` (ou `calloc`) e `free`, declaradas em `stdlib.h`.

`malloc` recebe um tamanho em bytes, aloca uma região de memória com esse tamanho, e retorna um ponteiro para o começo dessa região, ou `NULL` em caso de erro. Essa é uma forma válida de fazer o que tentamos no exemplo anterior:

```
size_t size;
/* leia size de algum lugar */
int *p = malloc(size * sizeof(int)); /* espaço para size ints */
if(p == NULL) {
    /* tratamento de erro */
}
```

Há também a função `calloc`, que recebe dois parâmetros: o número de elementos e o tamanho de cada elemento, e também retorna uma região de memória com o tamanho necessário - mas os bytes dessa região são garantidamente inicializados para `0`, o que livra o programador da possibilidade de problemas relacionados a memória não-inicializada. O exemplo anterior, usando `calloc`:

```
size_t size;
/* leia size de algum lugar */
int *p = calloc(size, sizeof(int));
if(p == NULL) {
    /* tratamento de erro */
}
```

Após uma chamada a qualquer uma das duas funções, a memória alocada deve ser liberada pela função `free`:

```
int *p = calloc(size, sizeof(int));
/* ... */
free(p);
p = NULL; /* boas práticas; veja abaixo */
```

Alguns erros podem surgir pelo uso indevido de `free`:

- caso `free` nunca seja chamada para uma região de memória alocada, temos um memory leak; a memória só será recuperada pelo Sistema Operacional quando o processo terminar.
- caso a região alocada seja acessada depois de uma chamada a `free`, temos um caso de use-after-free, que pode ser explorado como falha de segurança. Para evitar isso, é prudente atribuir a `NULL` todo ponteiro apontando para aquela região depois de chamar `free`.
- caso `free` seja chamada duas vezes para a mesma região de memória, temos um double-free, que é um caso de corrupção de memória e normalmente faz com que o programa

termine.

Nota: o tamanho de uma região de memória alocada dinamicamente **não** pode ser obtido com `sizeof`:

```
int *p = malloc(num * sizeof(int));  
printf("%d", sizeof(p));  
free(p);
```

Imprime o valor de `sizeof(int*)`, independentemente de quantos bytes foram alocados por `malloc`. Novamente, isso ocorre porque `p` é um ponteiro, não uma array.

Nota: é comum ver casts depois de uma chamada a `malloc` ou `calloc`:

```
int *p = (int*) malloc(num * sizeof(int));
```

Porém, esse cast não é necessário, pois `malloc` tem tipo de retorno `void*`, que pode ser implicitamente convertido para qualquer outro tipo de ponteiro (vide seção `void*`).

(Nomenclatura: há quem chame a região de memória alocada por `malloc` e similares de "array alocada dinamicamente", mas evitamos chamar isso de array, porque o que temos de fato é um ponteiro.)

Revision #1

Created Mon, Jan 28, 2019 11:35 PM by Luana

Updated Mon, Jan 28, 2019 11:35 PM by Luana