Ponteiros

Prof. John Lenon C. Gardenghi

Nesta aula, tratamos de duas aplicações de ponteiros:

1. ponteiros como conceito de vetores e
2. alocação dinâmica de memória.

2 Ponteiros e vetores

Retomando a revisão, em C, declaramos um vetor da seguinte forma

```
int vetor[tamanho];
```

Mas qual a definição conceitual de vetor? Há duas equivalentes.

- `vetor` é um vetor de `tamanho` posições, cada uma das posições são armazenadas contiguamente na memória e
- `vetor` é um ponteiro que aponta para a primeira posição do vetor na memória.

Portanto, se fizermos

```
int *bPtr;
bPtr = b;
```

podemos operar tanto sobre `b` quanto `bPtr`.

Exemplo 1. *Implemente o programa do Slide 6 e observe as saídas. O que podemos observar?*

Exemplo 2. *A função do Slide 7 promete devolver os três primeiros números primos maiores que 1000. Onde está o erro?*

Neste contexto, ponteiros são operadores válidos em operações de comparação e aritméticas. Um ponteiro pode ser incrementado (`++`) ou decrementado (`--`) e um inteiro pode ser adicionado a (`+` ou `+=`) ou subtraído de (`-` ou `-=`) um ponteiro.

Para explicitar o que é feito na aritmética de ponteiros: suponha que um ponteiro

```
int *ptr;
```

possua o endereço 3000 armazenado. Se fizermos `ptr + 2`, o resultado usual seria 3002. Entretanto, quando um ponteiro é incrementado ou decrementado por um inteiro, o endereço de memória não é apenas alterado pelo inteiro, mas sim pelo inteiro vezes o tamanho do dados que o ponteiro referencia. Em nosso exemplo, `ptr + 2 = 3008`, já que um inteiro ocupa 4 bytes em memória. O mesmo vale para subtração, incremento e decremento unários. Ou seja, a operação que é feita de fato é

```
ptr + i = ptr + i * sizeof(tipo_ptr);
```

Por outro lado, se tivermos

```
int *ptr1, *ptr2;
```

e `ptr1` possui o endereço 3000 e `ptr2` possui o endereço 3008, então

```
int x = ptr2 - ptr1 = ( 3008 - 3000 ) / 4 = 2.
```

3 Alocação dinâmica de memória

Uma aplicação importante de ponteiro é a alocação dinâmica de memória.

A função `malloc` (o nome é uma abreviatura de *memory allocation*) aloca espaço para um bloco de bytes consecutivos na memória RAM do computador e devolve o endereço base desse bloco. O número de bytes é especificado no argumento da função. No seguinte fragmento de código, `malloc` aloca 1 byte:

```
char *ptr;
ptr = malloc(1);
scanf ("%c", ptr);
```

Para alocar espaço para outros tipos de variável, cujo tamanho podemos não saber de cabeça, podemos usar o operador `sizeof`. Por exemplo, para alocar espaço para um `double`,

```
double *ptr;
ptr = malloc( sizeof(double) );
scanf( "%lf", ptr );
```

A função `malloc` retorna um ponteiro de `void`. Se a alocação de memória não der certo por algum motivo, ela retorna `NULL`. Por isso, sempre que fizermos alocação dinâmica de memória, é necessário fazer a verificação.

```
double *ptr;
ptr = malloc( sizeof(double) );
if ( ptr == NULL ) {
  scanf( "%lf", ptr );
}
```

Alocação de memória transfere para o programador a responsabilidade por aquele espaço que foi alocado. Por isso, é necessário **liberar** este espaço ao final do seu uso. Para isso, existe a função `free`. Para liberar o espaço alocado para `ptr` nos exemplos acima, basta fazer

```
free(ptr);
```

Observações

- Para fazer uso das funções de alocação dinâmica de memória, é necessário incluir a biblioteca `stdlib.h`.
- `NULL` é uma constante da `stdlib.h` que vale zero.
- O protótipo de `malloc` é

  ```
  void *malloc(size_t size);
  ```

  Aqui cabem duas observações importantes.

  – A função `malloc` retorna um ponteiro para void. Por isso, é comum fazer um *cast* na hora da alocação dinâmica. Por exemplo,

  ```
  int *ptr = ( int * ) malloc( 5*sizeof(int) );
  ```

  Todavia, o *cast* é altamente não recomendável, pois o ponteiro de `void` é automaticamente convertido para outro tipo de ponteiro.

  – `size_t` é um tipo especial de inteiro sem sinal que cabe todo valor possível para índice de um vetor. É um tipo da biblioteca `stddef.h`, que é incluída pela `stdlib.h`.

Como todas as vezes precisamos fazer a verificação se a alocação dinâmica deu certo, podemos fazer, em nossos programas, as chamadas *funções embalagens* (do inglês, *wrapper functions*). Para o `malloc`, por exemplo, podemos fazer

```
void *mallocc( size_t nbytes ) {
  void *ptr;
  ptr = malloc( nbytes );
  if ( ptr == NULL ) {
    printf( "Erro de alocacao de memoria.\n" );
    exit( EXIT_FAILURE );
  }
  return ptr;
}
```

e chamar `mallocc` sempre que alocarmos um vetor.

Observação: a função `exit` vem da biblioteca `stdlib` e interrompe a execução do programa e fecha todos os arquivos que o programa tenha porventura aberto. Se o argumento da função for 0, o sistema operacional é informado de que o programa terminou com sucesso; caso contrário, o sistema operacional é informado de que o programa terminou de maneira excepcional. As constantes `EXIT_SUCCESS` e `EXIT_FAILURE` valem 0 e 1, respectivamente, e estão declaradas na biblioteca `stdlib.h`. Chamar `exit (XXX)` de alguma função equivale a chamar `return XXX` do main.