ESTRUTURA DE DADOS

PONTEIROS

Profa. Fabrícia Damando Santos
fabriciadamando@gmail.com

Estruturas e ponteiros

- Como ocorre com as variáveis, as estruturas também podem ser referenciadas por ponteiros.

- Assim, definindo-se por exemplo o ponteiro *p para a estrutura apresentada (lapis), pode-se usar a sintaxe (*p).dureza.

- Porém, para referenciar o ponteiro há ainda outra sintaxe, através do operador -> , como por exemplo, p->dureza.

Ponteiros para estruturas

- Ponteiros para estruturas:
- Acesso ao valor de um campo x de uma variável estrutura p: p.x
- Acesso ao valor de um campo x de uma variável ponteiro pp: pp->x

```
struct ponto *pp; /* formas equivalentes de acessar o valor de um campo x */
(*pp).x = 12.0;
pp->x = 12.0;
```

Struct – com função


```
/* função que imprima as coordenadas do ponto */

void imprime (struct ponto p)
{
printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x,
   p.y);
}
```

Struct – função – passagem de parâmetros

- Passagem de estruturas para funções:
  - Referência:
    - apenas o ponteiro da estrutura é passado, mesmo que não seja necessário alterar os valores dos campos dentro da função
    - Ex:

```
/* função que imprima as coordenadas do
     ponto */
void imprime (struct ponto* pp)
{
    printf("O ponto fornecido foi: (%.2f,%.2f)\n",
     pp->x, pp->y);
}

void captura (struct ponto* pp)
{
printf("Digite as coordenadas do ponto(x y): ");
scanf("%f %f", &p->x, &p->y);
}
int main (void)
{
struct ponto p;
captura(&p);
imprime(&p);
return 0;
 }
```

Alocação dinâmica de memória

- A alocação é *estática*
  - acontece antes que o programa comece a ser executado:
  - char c;
  - int i;
  - int v[10];
- alocação *dinâmica* de memória
  - a quantidade de memória a alocar só se torna conhecida durante a *execução* do programa
  - ***Funções***: malloc, reallo c e free,
  - ***Biblioteca:*** stdlib.

Função malloc

- A função malloc (*memory allocation*) aloca um bloco de bytes consecutivos na memória RAM e devolve o endereço desse bloco
- O número de bytes é especificado no argumento da função
  - EX: alicação de 1 byte
    - char *ptr;
    - ptr = **malloc** (1);
    - scanf ("%c", ptr);

- Função malloc:
  - Para alocar um objeto que ocupa mais de 1 byte, é preciso recorrer ao operador sizeof, que diz quantos bytes o tipo de objeto desejado tem

```
typedef struct {
 int dia, mes, ano;
} data;
data *d;
d = malloc (sizeof (data));
d->dia = 31;
d->mes = 12;
d->ano = 2014;
```

Os parênteses na expressão sizeof (data) são necessários porque data é um tipo-de-dados.

Struct -alocação dinâmica

- Revisando....
  - tamanho do espaço de memória alocado dinamicamente é dado pelo operador **sizeof** aplicado sobre o tipo estrutura
  - função **malloc** retorna o endereço do espaço alocado, que é então convertido para o tipo ponteiro da estrutura
  - Ex:

```
struct ponto* p;

p = (struct ponto*) malloc (sizeof(struct ponto));
...
p->x = 12.0;
...
```

Função free

- As variáveis alocadas dinamicamente continuam a existir mesmo depois que a execução da função termina.
- Se for necessário liberar a memória ocupada por essas variáveis, é preciso recorrer à função free.

- Ex:

```
(free(ptr);
ptr = NULL;
```

Aplicação de ponteiros

- Suponha que precisamos de uma função que troque os valores de duas variáveis inteiras, digamos i e j.
- É claro que a função

```
void troca (int i, int j) { // errado!
int temp;
temp = i;
i = j;
j = temp; }
```

- não produz o efeito desejado, pois recebe apenas os *valores* das variáveis e não as variáveis propriamente ditas.
- A função recebe cópias das variáveis e troca os valores dessas cópias, enquanto as variáveis originais permanecem inalteradas.
- Para obter o efeito desejado, *é preciso passar à função os endereços das variáveis:*

Função com passagem de parâmetro Endereços das variáveis

```
void troca (int *p, int *q)
  {
  int temp;
  temp = *p;
  *p = *q;
  *q = temp; }
```

- Para aplicar essa função às variáveis i e j basta dizer

```
troca (&i, &j);
```

- ou talvez

```
int *p, *q;
p = &i;
q = &j;
troca (p, q);
```

Revisão

- A memória RAM de qualquer computador é uma sequência de bytes. Cada byte armazena um de 256 possíveis valores. Os bytes são numerados sequencialmente e o número de um byte é o seu *endereço* (= *address*).
- Cada objeto na memória do computador ocupa um certo número de bytes consecutivos.
  - Char ocupa 1 byte.
  - Int ocupa 4 bytes
  - Double ocupa 8 bytes em muitos computadores.
- O número exato de bytes de um objeto é dado pelo operador **sizeof**: a expressão sizeof (int), por exemplo, dá o número de bytes de um int no seu computador.

- Cada objeto na memória tem um *endereço*.
- Na maioria dos computadores, o endereço de um objeto é o endereço do seu primeiro byte.
- Por exemplo, depois das declarações

```
char c;
int i;
struct {
    int x, y;
} ponto;
int v[4];
```

Os endereços das variáveis poderiam ser os seguintes:

c *89421*
i *89422*
ponto *89426*
v[0] *89434*
v[1] *89438*
v[2] *89442*

Exercício 1

- Considere um cadastro de produtos de um estoque, com as seguintes informações para cada produto:
  - Código de identificação do produto: representado por um valor inteiro
  - Quantidade disponível no estoque: representado por um número inteiro
  - Preço de venda: representado por um valor real

- Questões:
  - Defina uma estrutura em C, denominada produto, que tenha os campos apropriados para guardar as informações de um produto, conforme descrito acima.

  - Escreva uma função que receba os dados de um produto (código, quantidade e preço) e retorne o endereço de um struct produto criado dinamicamente e inicializado com os valores recebidos como parâmetros pela função. Essa função deve ter o seguinte protótipo:

    - struct produto* cria (int cod, char* nome, int quant, float preco);