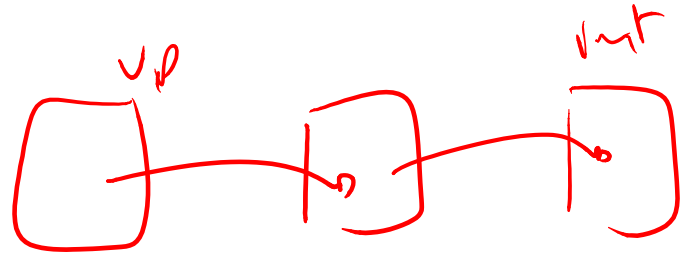
Introdução à Programação C

Fabio Mascarenhas - 2014.2

http://www.dcc.ufrj.br/~fabiom/introc

Vetores de ponteiros

- Podemos também ter vetores de qualquer tipo, inclusive ponteiros
- A sintaxe é a mesma para outros tipos de vetor: `int *vp[]` é um vetor de ponteiros para inteiros



- Também podemos usar `int **vp`, ou `int** vp`
- A principal utilidade disso é poder declarar vetores de cadeias de caracteres, já que `char vs[][]` não é um tipo válido, mas `char *vs[]` é
- Em um vetor de cadeias de caracteres, cada membro do vetor pode ter uma *capacidade* diferente (capacidade é quanto espaço para caracteres a cadeia tem, versus o tamanho que é quantos caracteres ela realmente tem)

Alocando um vetor de ponteiros

- Para alocar dinamicamente um vetor de ponteiros, primeiro precisamos alocar espaço para os ponteiros, depois alocar o espaço para onde cada ponteiro irá apontar

```
/* aloca um vetor de n vetores para números
inteiros, onde o tamanho de cada vetor é dado
pelo vetor ts */
static int** aloca_vetores(int ts[], int n) {
    int **vv = malloc(n * sizeof(int*));
    int i;
    for(i = 0; i < n; i++) {
        vv[i] = malloc(ts[i] * sizeof(int));
    }
    return vv;
}
```

```
/* libera um vetor de n vetores para
inteiros */
static void libera_vetores(int *vv[], int n)
{
    int i;
    for(i = 0; i < n; i++) {
        free(vv[i]);
    }
    free(vv);
}
```

Vetores dinâmicos

- Usando alocação dinâmica não precisamos ficar restritos a vetores de tamanho fixo

- Podemos representar um *vetor dinâmico* com três partes:

  - Uma *capacidade* (quantos elementos cabem no vetor atual)

  - Um *tamanho utilizado* (quantos elementos efetivamente estão no vetor)

  - O vetor em si

- A ideia é alocar um novo vetor toda vez que acaba o espaço, liberando o antigo

Desenhando retângulos

- Como exemplo de vetores dinâmicos, vamos criar um programa em que o usuário pode por retângulos na tela com o botão esquerdo do mouse, e apagá-los com o botão direito

- Todos os retângulos têm o mesmo tamanho, e cores aleatórias

- Temos então cinco vetores dinâmicos, para as coordenadas do retângulo e seus componentes de cor

Dados Estruturados

- Vários dados com os quais trabalhamos são *estruturados*, ou compostos por partes
  - Uma cor é composta por três componentes verde, vermelho e azul
  - Um ponto cartesiano é composto de duas coordenadas
  - Um tijolo no *Breakout* é composto de uma posição, uma cor e se está visível ou não
- Atualmente usamos várias variáveis para representar esses dados, mas podemos agrupar os diferentes pedaços em uma única variável com um *tipo estrutura*

Tipo Estrutura

- Uma estrutura ou struct é um tipo de dado composto por *campos*, cada qual possui seu próprio tipo e um nome que o identifica

- *Declaramos* um tipo estrutura com uma declaração `struct`:

```
struct Cor {
    double r;
    double g;
    double b;
};
```

- Declaramos variável com um tipo struct também usando a palavra chave `struct`, e o nome que demos:

```
struct Cor c;
```

Inicializando

- Podemos inicializar uma estrutura do mesmo modo que inicializamos um vetor:

```
struct Cor c = { 0.5, 0.0, 0.3 };
```

- Os campos na inicialização são dados na mesma ordem em que foram declarados

- Também podemos inicializar uma estrutura atribuindo diretamente aos campos, usando o operador `.` (ponto):

```
c.r = 0.5;
c.g = 0.0;
c.b = 0.3;
```

Acessando campos

- Podemos usar o mesmo operador . para ler campos de uma estrutura, não apenas para escrever:

```
tela_ret(100, 100, 50, 100, c.r, c.g, c.b);
```

- Também podemos obter o endereço de um campo de uma estrutura:

```
cor_aleat(&c.r, &c.g, &c.b);
```

- Podemos também obter um ponteiro para a estrutura inteira! Isso vai ser útil para passar para e retornar estruturas de funções

Passando (ponteiros para) estruturas

- Embora seja possível passar uma estrutura diretamente para uma função, o recomendável é sempre passar e retornar *ponteiros* para estruturas, por razões de eficiência:

```
void retangulo(int x, int y, int larg, int alt, struct Cor *c) {
    tela_ret(x, y, larg, alt, (*c).r, (*c).g, (*c).b);
}
```

- Trabalhar com ponteiros para estruturas é tão comum que existe sintaxe especial para acessar o campo de uma estrutura via um ponteiro:

```
void retangulo(int x, int y, int larg, int alt, struct Cor *c) {
    tela_ret(x, y, larg, alt, c->r, c->g, c->b);
}
```

Retornando (ponteiros para) estruturas

- Uma estrutura pode ser alocada dinamicamente com `malloc`, e assim podemos retornar um ponteiro para ela:

```
struct Cor *cor_aleat() {
    struct Cor *c = malloc(sizeof(struct Cor));
    c->r = rand() * 1.0 / RAND_MAX;
    c->g = rand() * 1.0 / RAND_MAX;
    c->b = rand() * 1.0 / RAND_MAX;
    return c;
}
```

- Naturalmente precisamos liberar a memória usando `free` depois de terminar nosso uso dela

Aninhamento de estruturas

- Uma estrutura pode ter outras estruturas como campos

- Um tijolo em nosso jogo Breakout é composto de um ponto dizendo onde ele está, de uma cor e de um flag dizendo se está visível ou não

```
struct Ponto {
    double x;
    double y
};

struct Tijolo {
    struct Ponto pos;
    struct Cor cor;
    int visivel;
};
```

```
struct Cor {
    double r;
    double g;
    double b;
};
```

- O acesso é feito por encadeamento: se `t` é um ponteiro pra um tijolo, `t->cor.g` dá o componente verde de sua cor, e `t->pos.x` a coordenada `x` de sua posição

Vetores de Estruturas

- Podemos ter vetores de estruturas, do mesmo modo que vetores de outros tipos

- Esses vetores podem tanto ser alocados estaticamente, dando um tamanho na declaração, ou dinamicamente, com malloc

- Vamos revisitar o “desenhando retângulos” do slide 5 para ter apenas um vetor que usa uma estrutura para representar os retângulos, ao invés de 5