Introdução à Programação C

Fabio Mascarenhas - 2014.2

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

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

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

```
struct Tijolo t;
t.cor.g = 0.5;
t.pos.x = 10;

struct Tijolo*
```

- 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

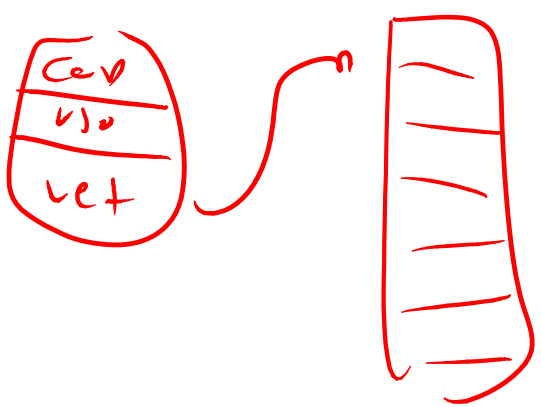
- Esses vetores podem tanto ser alocados estaticamente, dando um tamanho na declaração, ou dinamicamente, com malloc

- 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

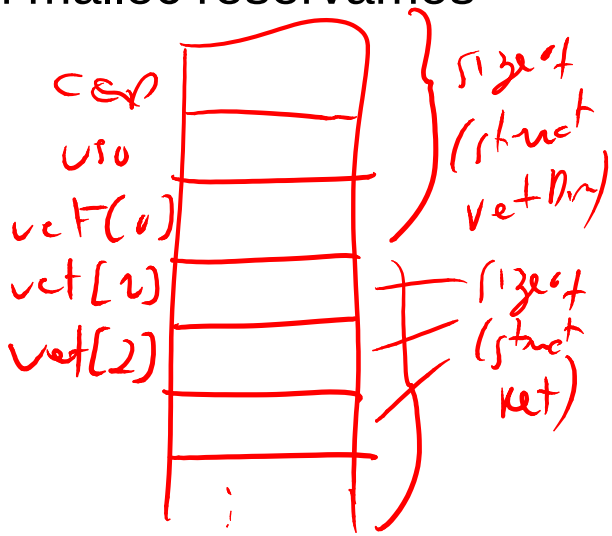
Estruturas com vetores

- Podemos ter um vetor com um tamanho fixo como campo de uma estrutura

- Há também um truque para ter vetor de tamanho variável como *último* campo de uma estrutura: declaramos como um vetor de apenas uma posição, e na hora de alocar espaço para a estrutura como um todo com malloc reservamos espaço para os elementos “extras”:




```
struct VetDin {
    int cap;
    int uso;
    struct Ret vet[1];
};
```




```
struct VetDin *vd = malloc(sizeof(struct VetDin) + (n-1) * sizeof(struct Ret));
```

Vetores dinâmicos de estruturas

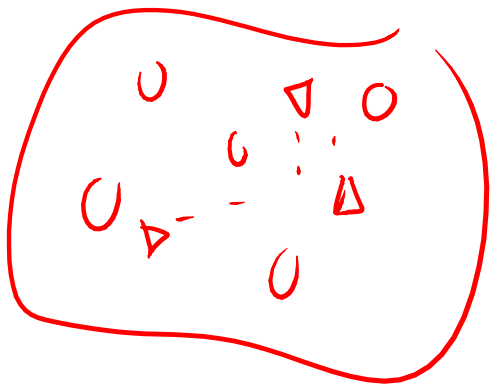
- Para juntar todos os tópicos da aula de hoje, vamos fazer uma última versão do programa “desenhando retângulos” do slide 5 que usa estruturas para representar tanto os retângulos como o “vetor dinâmico” de retângulos

Dados heterogêneos

- Nem sempre podemos descrever um tipo de dado composto com um único conjunto de campos

- Pense em um jogo com vários tipos de objetos na tela, todos eles em alguma posição, alguma cor, se movendo com determinada velocidade, e com um retângulo para determinar colisões entre eles, mas diferentes formas

- Vamos pensar em uma versão *multiplayer* do *Asteroids*, onde temos na tela asteróides, com tamanho, naves, com direção, número de vidas e um número que identifica o jogador, e tiros, com o número que identifica de qual nave eles saíram

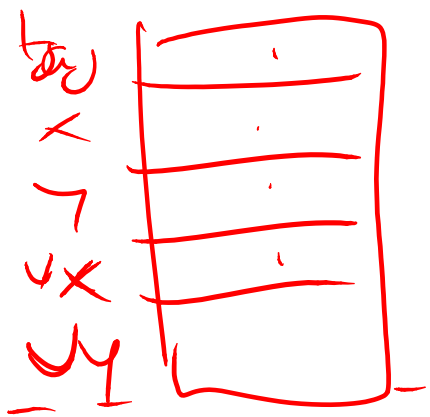


Compartilhando campos

- Uma estrutura na memória é uma sequência de caixas, uma para cada campo, *na ordem em que os campos aparecem*

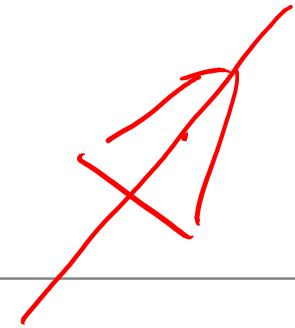
- Isso quer dizer que duas estruturas que começam com os mesmos campos têm a parte inicial de sua representação na memória com a mesma “forma”

```
struct Obj {
    char tag;
    double x;
    double y;
    double vx;
    double vy;
};
```



```
struct Ast {
    char tag;
    double x;
    double y;
    double vx;
    double vy;
    int tam;
};
```

Reinterpretando



- Se temos um ponteiro para uma `struct Ast`, então podemos reinterpretar ele como ponteiro para `struct Obj` e continuar acessando os campos de `Obj`:

```
struct Obj *po = (struct Obj*)past;
```

- Se sabemos que nosso ponteiro para uma `struct Obj` na verade aponta para uma `struct Ast` (via o campo `tag`, por exemplo), podemos fazer a operação inversa, e ter acesso aos campos exclusivos de `struct Ast`:

```
struct Ast *past = (struct Ast*)po;
```

Vetores de Obj

- Mas temos um problema se quisermos um vetor que misture asteróides, naves e tiros!

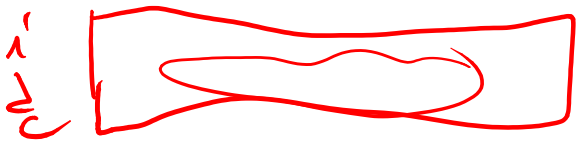
- Um vetor de `struct Obj` não vai funcionar, pois o tamanho de uma `struct Obj` é diferente (e menor) que o tamanho dessas outras estruturas, vai faltar espaço

- Uma solução é usar uma *união* para reservar espaço em `struct Obj` para os campos das outras estruturas

Uniões

- Uma união é um tipo de dado com campos como uma estrutura, mas na qual os campos *ocupam o mesmo local na memória*

- A ideia é que iremos acessar exclusivamente apenas um desses campos para cada instância de uma união



```
union Foo {
    int i;
    double d;
    char c;
};
```

```
union Foo f1;
f1.i = 10;
union Foo f2;
f2.d = 3.5;
union Foo f3;
f3.c = 'a';
```

- Só podemos acessar o campo `i` em `f1`, o campo `d` em `f2` e o campo `c` em `f3`

Reservando espaço em `struct Obj`

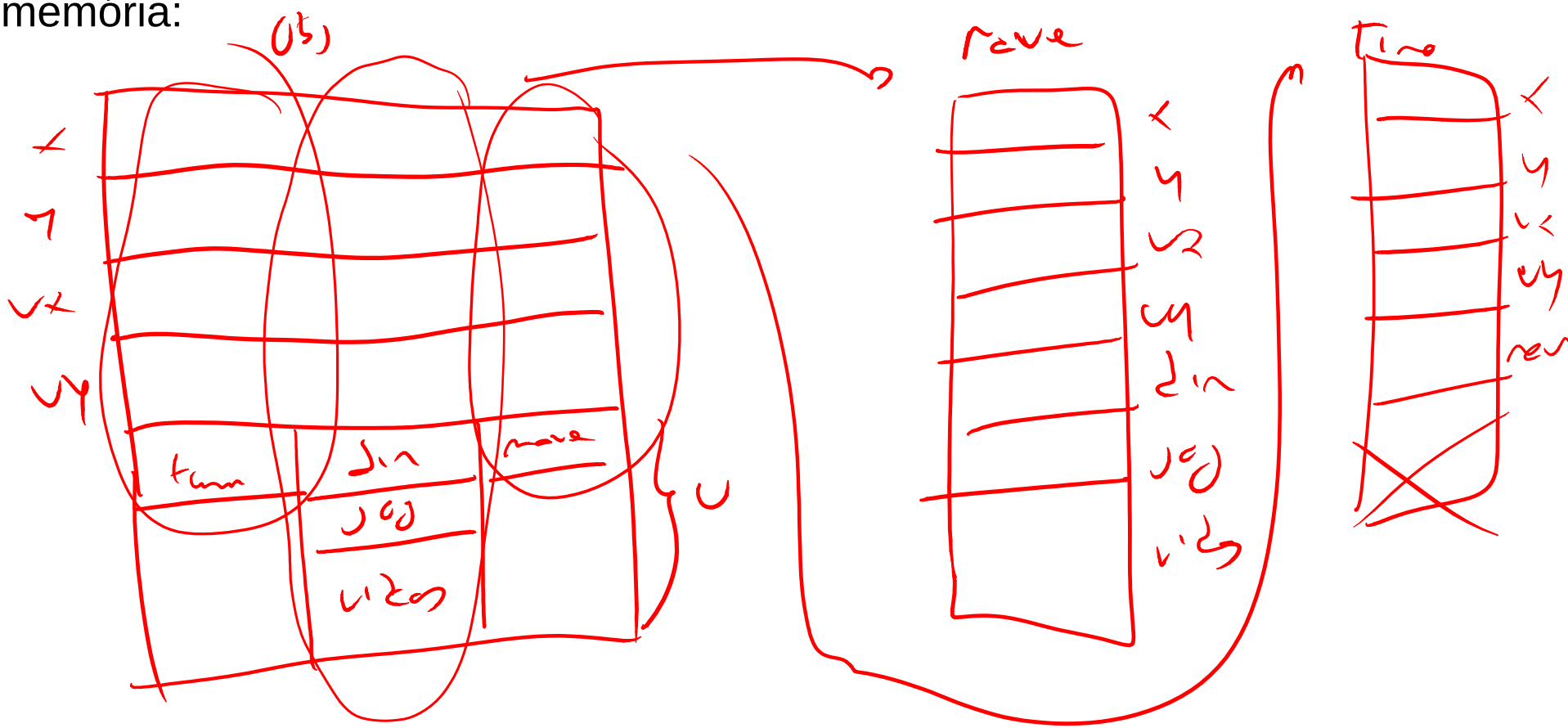
- Podemos usar uma união em que cada campo é uma estrutura com os campos extras que cada objeto do nosso jogo tem, e assim reservar o espaço necessário em `struct Obj`:

```
struct AstExtra {
    int tam;
};

struct NaveExtra {
    double dir;
    int jog;
    int vidas;
};

struct TiroExtra {
    int nave;
};
```

```
union Extras {
    struct AstExtra a;
    struct NaveExtra n;
    struct TiroExtra t;
};

struct Obj {
    double x;
    double y;
    double vx;
    double vy;
    union Extras u;
};
```

Formato de memória

- Podemos desenhar um esquema de como essas estruturas estão formatadas na memória:



Juntando as peças

- Agora que struct Obj tem espaço suficiente para qualquer de nosso objetos de jogo, podemos armazená-los todos em um mesmo vetor dinâmico

- Funções genéricas podem fazer o movimento dos objetos do jogo, e verificar suas colisões

- Para desenhar os objetos, usamos a tag e reinterpretação para despachar cada objeto para uma função específica que o desenha

- Usamos funções auxiliares para criar novos objetos e adicioná-los ao vetor dinâmico de objetos