

Tipos definidos pelo usuário

struct

Uma `struct` é um tipo composto por vários outros. Por exemplo:

```
struct cliente {  
    char *nome;  
    int idade;  
    char *endereco;  
    /* ... */  
};
```

Define o tipo `struct cliente` como uma composição dos tipos entre `{ }`, com os nomes acima. Cada sub-tipo da struct é chamado de *campo* da struct.

Podemos declarar uma variável desse tipo simplesmente como:

```
struct cliente c1;
```

Note que o nome do tipo é `struct cliente` e o da variável é `c1`.

Podemos também declarar variáveis de um tipo `struct` juntamente com a declaração do tipo:

```
struct ponto {  
    double x;  
    double y;  
    double z;  
} p1, p2, p3;
```

Declara as variáveis `p1`, `p2` e `p3` do tipo `struct ponto`. Depois disso, ainda podemos declarar mais variáveis desse tipo normalmente:

```
struct ponto p4, p5;
```

Se declararmos variáveis juntamente com a `struct` tipo de declaração, é possível omitir o nome da `struct`:

```
struct {  
    int x;  
    int y;  
} vetor1, vetor2;
```

Nesse caso temos uma struct anônima. Como esse tipo não tem nome, não é possível declarar nenhuma outra variável desse tipo além de `vetor1` e `vetor2`.

Podemos acessar cada um dos campos de um tipo `struct` com o operador `.`

```
p1.x = 1;  
p1.y = 2;  
p1.z = -3;  
printf("x = %f y = %f z = %f\n", p1.x, p1.y, p1.z);
```

Como em outras variáveis, é possível atribuir um valor a structs durante sua declaração, de um modo similar à atribuição de arrays:

```
/* atribui o primeiro campo da struct ao valor 1, o segundo ao valor 2 e o terceiro, a -3 */  
struct ponto p1 = {1, 2, -3};
```

Como com qualquer outro tipo, é possível declarar ponteiros para `struct`s, e é possível alocar memória para esses ponteiros dinamicamente:

```
struct ponto *ptr = malloc(sizeof(struct ponto));  
/* ... */  
free(ptr);  
ptr = NULL;
```

Nesse caso, poderíamos acessar o campo `x` desse ponto usando `(*ptr).x` (os parênteses são necessários por causa da precedência dos operadores). Porém, por conveniência, criou-se o operador `->`, de modo que podemos escrever `ptr->x`, o que é equivalente, por definição, à construção anterior.

Note que podemos alocar memória para vários pontos de maneira semelhante:

```
struct ponto *ptr = malloc(10 * sizeof(struct ponto)); /* 10 pontos */  
/* ... */  
free(ptr);  
ptr = NULL;
```

Nesse caso, ainda podemos acessar o campo `x` do primeiro ponto com `ptr->x`, mas para acessar esse mesmo campo de outro ponto (digamos, o quarto) teríamos que fazer `*(ptr+3).x`, o que

equivale a `ptr[3].x`.

Funções podem receber e/ou retornar `struct`s. Contudo, nos dois casos, gera-se uma cópia da `struct`, o que pode demandar muita memória se a struct tiver muitos campos. Por isso, costuma-se usar ponteiros para `struct` nesse caso. (Mas lembre-se que não podemos retornar ponteiros para variáveis locais!)

Uma `struct` pode conter um ponteiro para seu próprio tipo. Na verdade, essa é uma implementação comum de listas ligadas:

```
struct lista {
    int val; /* o valor desse elemento */
    struct lista *prox; /* ponteiro para o próximo elemento */
};
```

Mas como esse tipo pode declarar um ponteiro para si mesmo se a definição de `struct lista` ainda não terminou quando `prox` é declarado? A resposta tem a ver com tipos opacos.

Nota: declarações de structs são frequentemente acompanhadas de `typedef`, para que se possa omitir o nome `struct` ao usar o tipo. Por exemplo:

```
typedef struct ponto {
    int x;
    int y;
    int z;
} Ponto;
```

Nesse caso, `Ponto` se torna um outro nome para o tipo `struct ponto`. Porém, usar `typedef` com `struct`s é considerado má prática por vários guias de estilo, entre eles o do Linux, pois com uma quantidade suficientemente grande de tipos, pode-se esquecer que aquele tipo é uma struct e tratá-lo como um tipo primitivo.

Nota: embora C seja uma linguagem imperativa, `struct`s podem ser usadas juntamente com ponteiros de funções para emular a orientação a objetos, de maneira limitada.

union

Uma variável de um tipo `union` pode ter apenas um entre os tipos especificados em um dado momento. Esses tipos são definidos da mesma forma

```
union number {
    int i;
```

```
float f;

};
```

Variáveis de um tipo `union` podem ser declaradas das mesmas maneiras que variáveis de tipo `struct`.

Ao contrário das `struct`s, numa `union` apenas um dos campos é válido em um dado instante. Por exemplo:

```
union number num;

num.i = 5;
printf("%d\n", num.i); /* OK num.i é válido agora */
printf("%f\n", num.f); /* comportamento indefindo! */

num.f = 3.0;
printf("%f\n", num.f); /* OK, num.f é válido agora */
printf("%d\n", num.i); /* comportamento indefindo! */
```

Ou seja, se atribuírmos a um dos campos da `union`, qualquer tentativa de ler de qualquer outro campo gera um valor indefinido, mas a qualquer momento podemos atribuir um valor a outro campo.

Para saber qual campo da `union` é válido, costuma-se usar colocar a `union` dentro de uma `struct`, juntamente com uma `enum` que guarda o tipo:

```
struct numero {
    union {
        int i;
        float f;
    } valor; /* note que a union é anônima, valor é a variável, que é um campo da struct */
    enum {
        NUMERO_FLOAT,
        NUMERO_INT
    } tipo; /* enum também é anônima */
};
```

Por exemplo:

```
struct numero num;
num.valor.i = 2;
num.tipo = NUMERO_INT;
/* ... */
```

Nota: o tamanho de um tipo `union` é geralmente o tamanho do maior de seus campos.

enum

Um `enum` é um tipo que pode conter apenas alguns valores especificados. Por exemplo:

```
enum cor_semaforo {
    VERMELHO,
    AMARELO,
    VERDE
};
```

Qualquer variável do tipo `enum cor_semaforo` pode conter apenas um desses três valores. Cada valor da `enum` é internamente representado por um inteiro, e é possível atribuir valores explícitos a eles:

```
enum cor_semaforo {
    VERMELHO = 0,
    AMARELO = 1,
    VERDE = 2
};
```

Isso é um tanto parecido com

```
const int VERMELHO = 0;
const int AMARELO = 1;
const int VERDE = 2;
/* e então usamos int em vez de `enum cor_semaforo` */
```

Porém, usando-se a `enum`, há a garantia de que a variável só pode valer um desses três valores; o mesmo não ocorre quando se usa `int`.