

Estruturas de Dados

Módulo 8 – Tipos Estruturados

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO



Referências

Waldemar Celes, Renato Cerqueira, José Lucas Rangel,
Introdução a Estruturas de Dados, Editora Campus
(2004)

Capítulo 8 – Tipos estruturados

Tópicos

- Tipo estrutura
- Definição de novos tipos
- Aninhamento de estruturas
- Vetores de estruturas
- Tipo união
- Tipo enumeração

Tipo Estrutura

- **Motivação:**
 - manipulação de dados compostos ou estruturados
 - Exemplos:
 - ponto no espaço bidimensional
 - representado por duas coordenadas (x e y), mas tratado como um único objeto (ou tipo)
 - dados associados a aluno:
 - aluno representado pelo seu nome, número de matrícula, endereço, etc ., estruturados em um único objeto (ou tipo)

Ponto

X
Y

Aluno

Nome	
Matr	
End	Rua
	No
	Compl

Tipo Estrutura

- Tipo estrutura:
 - tipo de dado com campos compostos de tipos mais simples
 - elementos acessados através do operador de acesso “ponto” (.)

```
struct ponto          /* declara ponto do tipo struct */
{
  float x;
  float y;
};
...
struct ponto p;      /* declara p como variável do tipo struct ponto */
...
p.x = 10.0;          /* acessa os elementos de ponto */
p.y = 5.0;
```

Tipo Estrutura

```
/* Captura e imprime as coordenadas de um ponto qualquer */
#include <stdio.h>
struct ponto {
    float x;
    float y;
};
int main (void)
{
    struct ponto p;
    printf("Digite as coordenadas do ponto(x y): ");
    scanf("%f %f", &p.x, &p.y);
    printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
    return 0;
}
```

Basta escrever `&p.x` em lugar de `&(p.x)` .

O operador de acesso ao campo da estrutura tem precedência sobre o operador “endereço de”

Tipo Estrutura

- 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`
 - acesso ao *endereço* do campo x de uma variável ponteiro pp: `&pp->x`

```
struct ponto *pp;
```

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

```
pp->x = 12.0;
```

Tipo Estrutura

- Passagem de estruturas para funções – por valor:
 - análoga à passagem de variáveis simples
 - função recebe toda a estrutura como parâmetro:
 - função acessa a cópia da estrutura na pilha
 - função não altera os valores dos campos da estrutura original
 - operação pode ser custosa se a estrutura for muito grande

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

Tipo Estrutura

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

```
/* 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; }
```

Tipo Estrutura

- Alocação dinâmica de estruturas:
 - 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

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

Definição de Novos Tipos

- `typedef`
 - permite criar nomes de tipos
 - útil para abreviar nomes de tipos e para tratar tipos complexos

```
typedef unsigned char UChar;  
typedef int* PInt;  
typedef float Vetor[4];  
  
Vetor v; /* exemplo de declaração usando Vetor */  
...  
v[0] = 3;
```

- `UChar` o tipo char sem sinal
- `PInt` um tipo ponteiro para int
- `Vetor` um tipo que representa um vetor de quatro elementos

Definição de Novos Tipos

- `typedef`

- Exemplo: definição de nomes de tipos para as estruturas

```
struct ponto {  
    float x;  
    float y;  
};  
  
typedef struct ponto Ponto;  
typedef struct ponto *PPonto;
```

- `ponto` representa uma estrutura com 2 campos do tipo `float`
- `Ponto` representa a estrutura `ponto`
- `PPonto` representa o tipo ponteiro para a estrutura `Ponto`

Definição de Novos Tipos

- `typedef`
 - Exemplo: (definição utilizando um só `typedef`)

```
struct ponto {  
    float x;  
    float y;  
};  
  
typedef struct ponto Ponto, *PPonto;
```

- `ponto` representa uma estrutura com 2 campos do tipo `float`
- `Ponto` representa a estrutura `ponto`
- `PPonto` representa o tipo ponteiro para a estrutura `Ponto`

Definição de Novos Tipos

- `typedef`

- Exemplo: (definição em um comando só)

```
typedef struct ponto {  
    float x;  
    float y;  
} Ponto;
```

- `ponto` representa uma estrutura com 2 campos do tipo `float`
 - `Ponto` representa a estrutura `ponto`

Aninhamento de Estruturas

- Aninhamento de estruturas:
 - campos de uma estrutura podem ser outras estruturas
 - Exemplo:
 - definição de Círculo usando Ponto

```
struct circulo {  
    Ponto p;          /* centro do círculo */  
    float r;         /* raio do círculo */  
};  
  
typedef struct circulo Circulo;
```

```
/* Função para determinar se um ponto está ou não dentro de um círculo:  
   entrada: ponteiros para um círculo e para um ponto  
   saída:   1 = ponto dentro do círculo  
           0 = ponto fora do círculo
```

```
*/  
int interior (Circulo* c, Ponto* p)  
{  
    float d = distancia(&c->p, p);  
    return (d < c->r);  
}
```

&c->p : ponteiro para centro de c

p : ponteiro para o ponto

c->r : raio do círculo

d < c->r : testa se d é menor do raio

```
/* Função para a calcular distância entre 2 pontos:  
   entrada: ponteiros para os pontos  
   saída:   distância correspondente
```

```
*/  
float distancia (Ponto* p, Ponto* q)  
{  
    float d = sqrt((q->x - p->x)*(q->x - p->x) + (q->y - p->y)*(q->y - p->y));  
    return d;  
}
```

cálculo da distância:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

sqrt da biblioteca math.h

```

#include <stdio.h>
#include <math.h>
typedef struct ponto {
    float x;
    float y;
} Ponto;

typedef struct circulo {
    Ponto p;          /* centro do círculo */
    float r;         /* raio do círculo */
} Circulo;

int main (void)
{
    Circulo c;
    Ponto p;
    printf("Digite as coordenadas do centro e o raio do circulo:\n");
    scanf("%f %f %f", &c.p.x, &c.p.y, &c.r);
    printf("Digite as coordenadas do ponto:\n");
    scanf("%f %f", &p.x, &p.y);
    printf("Pertence ao interior = %d\n", interior(&c,&p));
    return 0;
}

```

Vetores de Estruturas

- Exemplo:
 - função para calcular o centro geométrico de conjunto de pontos
 - entrada: vetor de estruturas definindo o conjunto de pontos
 - saída: centro geométrico, dado por:

$$\bar{x} = \frac{\sum x_i}{n} \quad \bar{y} = \frac{\sum y_i}{n}$$

Vetores de Estruturas

```
Ponto centro_geom (int n, Ponto* v)
{
    int i;
    Ponto p = {0.0f, 0.0f}; /* declara e inicializa ponto */
    for (i=0; i<n; i++)
    {
        p.x += v[i].x;
        p.y += v[i].y;
    }
    p.x /= n;
    p.y /= n;
    return p;
}
```

– função retornando estrutura:

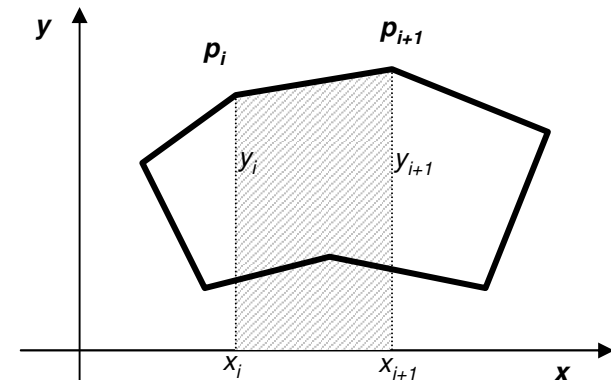
- para estruturas pequenas, este recurso facilita o uso da função
- para estruturas grandes, a cópia do valor de retorno pode ser caro

Vetores de Estruturas

- Exemplo:
 - função para calcular a área de um polígono plano delimitado por uma seqüência de n pontos
 - a área do polígono é a soma das áreas dos trapézios formados pelos lados do polígono e o eixo x
 - a área do trapézio definido pela aresta que vai do ponto p_i ao ponto p_{i+1} é dada por:

$$a = (x_{i+1} - x_i)(y_{i+1} + y_i)/2$$

- algumas “áreas” são negativas
- as áreas externas ao polígono são anuladas
- se a seqüência de pontos do polígono for dada em sentido anti-horário, a “área” terá valor negativo e a área do polígono é o valor absoluto do resultado da soma.



Vetores de Estruturas

```
float area (int n, Ponto* p)
{
    int i, j;
    float a = 0;
    for (i=0; i<n; i++)
    {
        j = (i+1) % n;          /* próximo índice (incremento circular) */
        a += (p[ j ].x - p[ i ].x)*(p[ i ].y + p[ j ].y)/2;
    }
    return fabs(a);
}
```

fabs

- função definida em [math.h](#)
- retorna o valor absoluto de um valor real

Vetores de Ponteiros para Estruturas

- Exemplo:
 - tabela com dados de alunos, organizada em um vetor
 - dados de cada aluno:

matrícula:	número inteiro
nome:	cadeia com até 80 caracteres
endereço:	cadeia com até 120 caracteres
telefone:	cadeia com até 20 caracteres

- Solução 1:
 - Aluno
 - estrutura ocupando pelo menos $4+81+121+21 = 227$ Bytes
 - tab
 - vetor de Aluno
 - representa um desperdício significativo de memória, se o número de alunos bem inferior ao máximo estimado

```
struct aluno {
    int mat;
    char nome[81];
    char end[121];
    char tel[21];
};

typedef struct aluno Aluno;

#define MAX 100
Aluno tab[MAX];
```

- Solução 2 (usada no que se segue):
 - tab
 - vetor de ponteiros para `Aluno`
 - elemento do vetor ocupa espaço de um ponteiro
 - alocação dos dados de um aluno no vetor:
 - nova cópia da estrutura `Aluno` é alocada dinamicamente
 - endereço da cópia é armazenada no vetor de ponteiros
 - posição vazia do vetor: **valor é o ponteiro nulo**

```
struct aluno {
    int mat;
    char nome[81];
    char end[121];
    char tel[21];
};
typedef struct aluno Aluno;
#define MAX 100
Aluno* tab[MAX];
```

- Inicializa - função para inicializar a tabela:
 - recebe um vetor de ponteiros
(parâmetro deve ser do tipo “ponteiro para ponteiro”)
 - atribui NULL a todos os elementos da tabela

```
void inicializa (int n, Aluno** tab)
{
    int i;
    for (i=0; i<n; i++)
        tab[i] = NULL;
}
```

- Preenche - função para armazenar novo aluno na tabela:
 - recebe a posição onde os dados serão armazenados
 - dados são fornecidos via teclado
 - se a posição da tabela estiver vazia, função aloca nova estrutura
 - caso contrário, função atualiza a estrutura já apontada pelo ponteiro

```
void preenche (int n, Aluno** tab, int i)
{
    if (i<0 || i>=n) {
        printf("Indice fora do limite do vetor\n");
        exit(1); /* aborta o programa */
    }
    if (tab[i]==NULL)
        tab[i] = (Aluno*)malloc(sizeof(Aluno));
    printf("Entre com a matricula:");
    scanf("%d", &tab[i]->mat);
    ...
}
```

- Retira - função para remover os dados de um aluno da tabela:
 - recebe a posição da tabela a ser liberada
 - libera espaço de memória utilizado para os dados do aluno

```
void retira (int n, Aluno** tab, int i)
{
    if (i<0 || i>=n) {
        printf("Indice fora do limite do vetor\n");
        exit(1); /* aborta o programa */
    }

    if (tab[i] != NULL)
    {
        free(tab[i]);
        tab[i] = NULL; /* indica que na posição não mais existe dado */
    }
}
```

- Imprimi - função para imprimir os dados de um aluno da tabela:
 - recebe a posição da tabela a ser impressa

```
void imprime (int n, Aluno** tab, int i)
{
    if (i<0 || i>=n) {
        printf("Indice fora do limite do vetor\n");
        exit(1); /* aborta o programa */
    }

    if (tab[i] != NULL)
    {
        printf("Matrícula: %d\n", tab[i]->mat);
        printf("Nome: %s\n", tab[i]->nome);
        printf("Endereço: %s\n", tab[i]->end);
        printf("Telefone: %s\n", tab[i]->tel);
    }
}
```

- `Imprimi_tudo` - função para imprimir todos os dados da tabela:
 - recebe o tamanho da tabela e a própria tabela

```
void imprime_tudo (int n, Aluno** tab)
{
    int i;
    for (i=0; i<n; i++)
        imprime(n,tab,i);
}
```

- Programa de teste

```
#include <stdio.h>

int main (void)
{
    Aluno* tab[10];
    inicializa(10,tab);
    preenche(10,tab,0);
    preenche(10,tab,1);
    preenche(10,tab,2);
    imprime_tudo(10,tab);
    retira(10,tab,0);
    retira(10,tab,1);
    retira(10,tab,2);
    return 0;
}
```

Tipo União

- union

- localização de memória compartilhada por diferentes variáveis, que podem ser de tipos diferentes
- uniões usadas para armazenar valores heterogêneos em um mesmo espaço de memória

```
union exemplo
{
    int i;
    char c;
}

union exemplo v;
```

- não declara nenhuma variável
- apenas define o tipo união

- campos i e c compartilham o mesmo espaço de memória
- variável v ocupa pelo menos o espaço necessário para armazenar o maior de seus campos (um inteiro, no caso)

Tipo União

- union

- acesso aos campos:

- operador ponto (.) para acessar os campos diretamente
 - operador seta (->) para acessar os campos através de ponteiro

```
union exemplo
{
    int i;
    char c;
}
union exemplo v;

v.i = 10;          /* alternativa 1 */
v.c = 'x';        /* alternativa 2 */
```

Tipo União

- union
 - armazenamento:
 - apenas um único elemento de uma união pode estar armazenado num determinado instante
 - a atribuição a um campo da união sobrescreve o valor anteriormente atribuído a qualquer outro campo

```
union exemplo
{ int i;
  char c;
}
union exemplo v;

v.i = 10;          /* alternativa 1 */
v.c = 'x';        /* alternativa 2 */
```

Tipo Enumeração

- `enum`
 - declara uma enumeração, ou seja, um conjunto de constantes inteiras com nomes que especifica os valores legais que uma variável daquele tipo pode ter
 - oferece uma forma mais elegante de organizar valores constantes

Tipo Enumeração

- Exemplo – tipo Booleano:

bool	declara as constantes FALSE e TRUE associa TRUE ao valor 1 e FALSE ao valor 0
Bool	declara um tipo cujos valores só podem ser TRUE (1) ou FALSE (0)
resultado	variável que pode receber apenas os valores TRUE ou FALSE

```
enum bool {  
    TRUE = 1,  
    FALSE = 0  
};  
  
typedef enum bool Bool;  
  
Bool resultado;
```

Resumo

```
struct      struct ponto { float x; float y; };

typedef    typedef struct ponto Ponto;
           typedef struct ponto *PPonto;

union      union exemplo { int i; char c; }
           union exemplo v;

enum      enum bool { TRUE = 1, FALSE = 0 };
           typedef enum bool Bool;
           Bool resultado;
```