

Programação de Computadores II

Cap. 8 – Tipos Estruturados

Livro: Waldemar Celes, Renato Cerqueira, José Lucas Rangel. *Introdução a Estruturas de Dados*, Editora Campus (2004)
Slides adaptados dos originais dos profs.: Marco Antonio Casanova e Marcelo Gattass (PUC-Rio)

1

Referências

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

Capítulo 8 – Tipos estruturados

2

Tópicos

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

3

Tipo Estrutura: Motivação

- 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

4

Tipo Estrutura: outra forma

- 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;
};
...
int main( ) {
    struct ponto p; /* declara p como variável do tipo struct ponto */
    ...
    p.x = 10.0; /* acessa os elementos de ponto */
    p.y = 5.0+p.x;
}
```

5

Tipo Estrutura: Exemplo

```
/* 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"

6

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

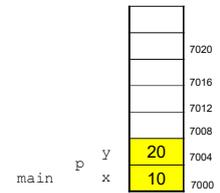
```
struct ponto p;
struct ponto *pp;
pp=&p;
...
(*pp).x = 12.0;      /* formas equivalentes de acessar o valor de um campo x */
pp->x = 12.0;
p.x = 12.0;
(&p)->x =12.0;
```

Qual o valor de...?

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

int main ( )
{
    struct ponto p = { 10,20};
    struct ponto *pp=&p;
    ...
}
```

Pilha de memória



Qual o valor de ...?

`p.y` ~~`pp.x`~~ `pp->x` `&(pp->y)`
`(&p)->x` `&(p.y)`

8

Passagem de estruturas por valor para funções

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

9

Estruturas como valor de retorno

```
/* Captura e imprime as coordenadas de um ponto qualquer */
#include <stdio.h>
struct ponto { float x; float y; };

struct ponto le( void){
    struct ponto tmp;
    printf("Digite as coordenadas do ponto(x y): ");
    scanf("%f %f", &tmp.x, &tmp.y);
    return tmp;
}

int main (void)
{
    struct ponto p=le();
    printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
    return 0;
}
```

Passagem de estruturas por referência para função

- 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", &pp->x, &pp->y);
}

int main (void)
{ struct ponto p; captura(&p); imprime(p); return 0; }
```

11

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;
...
free(p);
```

12

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

13

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**

14

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**

15

Definição de Novos Tipos – Boa Prática

- **typedef**

- Exemplo: (definição em um comando só)

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

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

```
main(void) {
    Ponto p1, p2, *p3;
    p3 = (Ponto *) malloc(sizeof(Ponto));
    ...
}
```

16

Aninhamento de Estruturas

- Aninhamento de estruturas (agregação):

- campos de uma estrutura podem ser outras estruturas

- Exemplo:

- definição de **Circulo** usando **Ponto**

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

typedef struct circulo Circulo;
```

17

Aninhamento de Estruturas

```
typedef struct circulo {
    Ponto p;
    float raio;
} Circulo;

// ----- main()
Circulo c;          Circulo *c2;  c2=&c;
c.p.x = 2;          c2->p.x = 2;
c.p.y = 4;          c2->p.y = 4;
c.raio = 6;         c2->raio = 6;

-----
typedef struct circulo {
    Ponto *p;
    float raio;
} Circulo;

// ----- main()
Circulo c;          Circulo *c2;  c2=&c;
c.p = (Ponto *) malloc(sizeof(Ponto));
c.p->x = 2;          c2->p->x = 2;
c.p->y = 4;          c2->p->y = 4;
c.raio = 6;         c2->raio = 6;
```

18

Aninhamento de Estruturas

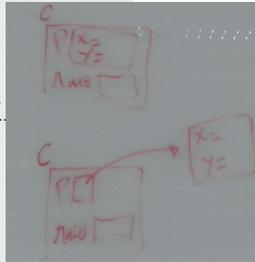
```
typedef struct circulo {
    Ponto p;
    float raio;
} Circulo;

// ----- main()
Circulo c;
c.p.x = 2;
c.p.y = 4;
c.raio = 6;

-----
typedef struct circulo {
    Ponto *p;
    float raio;
} Circulo;

// ----- main()
Circulo c;
c.p = (Ponto *) malloc(sizeof(Ponto));
c.p->x = 2;
c.p->y = 4;
c.raio = 6;

Circulo *c2;
c2->p.x = 2;
c2->p.y = 4;
c2->raio = 6;
```



19

```
/* Função para 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;
}

/* 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);
}
```

cálculo da distância:
 $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
 sqrt da biblioteca math.h

&c->p : ponteiro para centro de c

p : ponteiro para o ponto

20

```
#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 círculo:\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;
}
```

21

Chamada de função usando ponteiros

```
Circulo *c1;
Ponto *p1;
c1 = (Circulo *) malloc(sizeof(Circulo));
p1 = (Ponto *) malloc(sizeof(Ponto));

c1->p.x=10; c1->p.y=20; c1->r=30;
p1->x=15; p1->y=18;
interior(c1, p1);
```

22

Vetores de Estruturas

- Exemplo:

```
#define N 3

typedef struct funcionario{
    int mat;
    char nome[81];
    float salario;
} TFuncionario;

int main(void) {
    TFuncionario func[N]={{1,"func um",1000},{2,"func dois",2500},{3,"func tres",3000}};
    printf("\nNumero funcionarios com salario maior do que a media=%d\n",
        retNumFunc(func,N));
}
```

23

Vetores de Estruturas

```
int retNumFunc(TFuncionario *vet, int n) {
    int i;
    float mediaSalarios=0, numFuncionarios;
    for (i=0; i<n; i++)
        mediaSalarios = mediaSalarios + vet[i].salario;
    mediaSalarios=mediaSalarios/n;

    for (i=0; i<n; i++)
        if (vet[i].salario > mediaSalarios)
            numFuncionarios++;
    return numFuncionarios;
}
```

24

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

25

- 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 for 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];
```

26

- 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];
```

27

- 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;
}
```

28

- 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);
    ...
}
```

29

- 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 */
    }
}
```

30

- Imprime - 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("Índice 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);
  }
}
```

31

- Imprime_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);
}
```

32

- 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;
}
```

33

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)

34

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 */
```

35

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 */
```

36

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

37

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;
```

38

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;
```

39