Variáveis Compostas Heterogêneas

Algoritmos e Programação 2

Prof. Dr. Anderson Bessa da Costa
Universidade Federal de Mato Grosso do Sul

Introdução

- Até agora, variáveis armazenam apenas **um único tipo** de dado
  - Definido na declaração
- Variáveis servem, também, para representar entidades identificadas no problema real que será resolvido computacionalmente
- É bastante comum a necessidade de armazenar, dentro de uma mesma variável, **diferentes tipos** de dados
- É ai que entram em cena os **registros**

Registros

- **Registros** conseguem agregar vários dados acerca de uma mesma entidade
- Programadores podem gerar **novos tipos** de dados, não se limitando à utilização dos tipos de dados primitivos
- Cada dado contido em um registro é chamado **campo**
  - Os **campos** podem ser de diferentes tipos primitivos ou, ainda, podem representar outros **registros**
- É por essa razão que os registros são conhecidos, também, como **variáveis compostas heterogêneas**

Definição de Estrutura

- Os registros em C/C++, chamados de estruturas, são definidos por meio da utilização da palavra reservada **struct**, conforme apresentado a seguir

```
struct nome_da_estrutura {
    <tipo> <campo1>;
    <tipo> <campo2>;
    …
    <tipo> <campoN>;
};
```

Definição de Estrutura (Cont.)

- A partir da estrutura definida, o programa poderá considerar que existe um novo tipo e dado a ser utilizado chamado **nome_da_estrutura**
- Esse **novo tipo** de dado é capaz de armazenar várias informações cujos tipos podem ser diferentes

Exemplo 1

```
struct banco {
    int num;
    char titular[35];
    float saldo;
};
```

- Uma estrutura chamada **banco** foi definida. Isso significa que o programa poderá utilizar um **novo tipo** de dado
- Variáveis declaradas que fazem uso desse tipo poderão armazenar três valores: **num**, **titular** e **saldo**

Declaração de Estrutura

- Para que um programa em C/C++ utilize uma **struct**, é necessária a declaração de variáveis desse tipo, da seguinte forma:

```
struct nome_da_estrutura nome_da_variável;
```

- Considerando que estruturas representam **novos tipos** de dados, todas as operações e declarações realizadas com os tipos predefinidos da linguagem também poderão ser realizadas com as estruturas

Exemplo 2

```c
#include <stdio.h>
...
struct banco {
    int num;
    char titular[35];
    float saldo;
};
...
int main() {
    ...
    struct banco conta;
    ...
}
```

Representação Variável Conta

- Assim, conta tem três campos: **num**, **titular** e **saldo**
- A seguir, mostramos uma representação gráfica da variável **conta**:

Variável conta

num	
titular	
saldo	

Exemplo 3

```
#include <stdio.h>
...
struct banco {
    int num;
    char titular[35];
    float saldo;
};
...
int main() {
    ...
    struct banco contas[4][3];
    ...
}
```

- Observe que **contas** é uma matriz com 4 linhas e 3 colunas. Cada posição dessa matriz terá espaço para armazenar três valores: **num**, **titular** e **saldo**

Representação Variável Contas

	0		1		2	
0	num		num		num	
	titular		titular		titular	
	saldo		saldo		saldo	
1	num		num		num	
	titular		titular		titular	
	saldo		saldo		saldo	
2	num		num		num	
	titular		titular		titular	
	saldo		saldo		saldo	
3	num		num		num	
	titular		titular		titular	
	saldo		saldo		saldo	

Acesso a Campos da Estrutura

- O uso de uma **struct** é possível por meio do acesso individual a seus campos, quer seja para gravar quer seja para recuperar um dado
- O acesso a determinado campo da estrutura é feito informando-se o nome da variável, seguido por um ponto e pelo nome do campo desejado
- A forma geral a ser utilizada é descrita abaixo:

```
nome_da_variável_do_tipo_estrutura.nome_do_campo
```

Exemplo 4

```c
#include <stdio.h>
...
struct banco {
    int num;
    char titular[35];
    float saldo;
};
...
int main() {
    ...
    struct banco conta;
    ...
    printf("Digite o numero da conta: ");
    scanf("%d%*c", &conta.num);
    printf("Digite o nome do titular da conta: ");
    gets(conta.titular);
    printf("Digite o saldo da conta: ");
    scanf("%f%*c", &conta.saldo);
    ...
```

Exemplo 5

```c
#include <stdio.h>
...
struct empresa {
    char nome[50];
    float salario;
};
...
int main() {
    ...
    struct empresa funcionarios[4];
    ...
    for(i = 0; i < 4; i++) {
        printf("Digite o nome do funcionário %d : ", i);
        gets(funcionarios[i].nome);
        printf("Digite o salário do funcionário %d : ", i);
        scanf("%f%*c", &funcionarios[i].salario);
    }
    ...
}
```

Exemplo 5 (cont.)

- Cada posição desse vetor tem capacidade para armazenar os campos **nome** e **salário**, definidos na estrutura **empresa**
- Assim, como acontece com qualquer vetor, as posições devem ser acessadas com o uso de um índice

Representação Vetor Funcionários

0	nome	João
	salario	1000,00
1	nome	Maria
	salario	5000,00
2	nome	Pedro
	salario	1800,00
3	nome	Lúcia
	salario	2700,00

Exemplo 6

```
...
for(i = 0; i < 4; i++) {
    printf("\nFuncionario que ocupa a posicao %d no vetor:", i);
    printf("\nNome: %s", funcionarios[i].nome);
    printf("\nSalario: %6.2f", funcionarios[i].salario);
    printf("\n");
}
...
```

- A estrutura de repetição **for**, permite que as quatro posições do vetor sejam percorridas e os dados encontrados sejam mostrados

Exemplo 6 (cont.)

```
Funcionario que ocupa a posicao 0 no vetor:
Nome: Joao
Salario: 1000.00

Funcionario que ocupa a posicao 1 no vetor:
Nome: Maria
Salario: 5000.00

Funcionario que ocupa a posicao 2 no vetor:
Nome: Pedro 1800
Salario:   0.00

Funcionario que ocupa a posicao 3 no vetor:
Nome: Lucia
Salario: 2700.00
```

Palavra Reservada Typedef

- Declarações com **typedef** não produzem novos tipos de dados. Criam apenas novos nomes (sinônimos) para os tipos existentes

Exemplo 7

```c
#include <stdio.h>

typedef struct {
    char nome[20];
    char sobrenome[20];
    int idade;
    char sexo;
    double salario;
} funcionario ;

int main() {
    funcionario f = {“Marie”, "Silva", 30};

    printf("%s %s (%d anos)\n", f.nome, f.sobrenome, f.idade);

    return 0;
}
```

Palavra reservada

O nome deve vir para o final

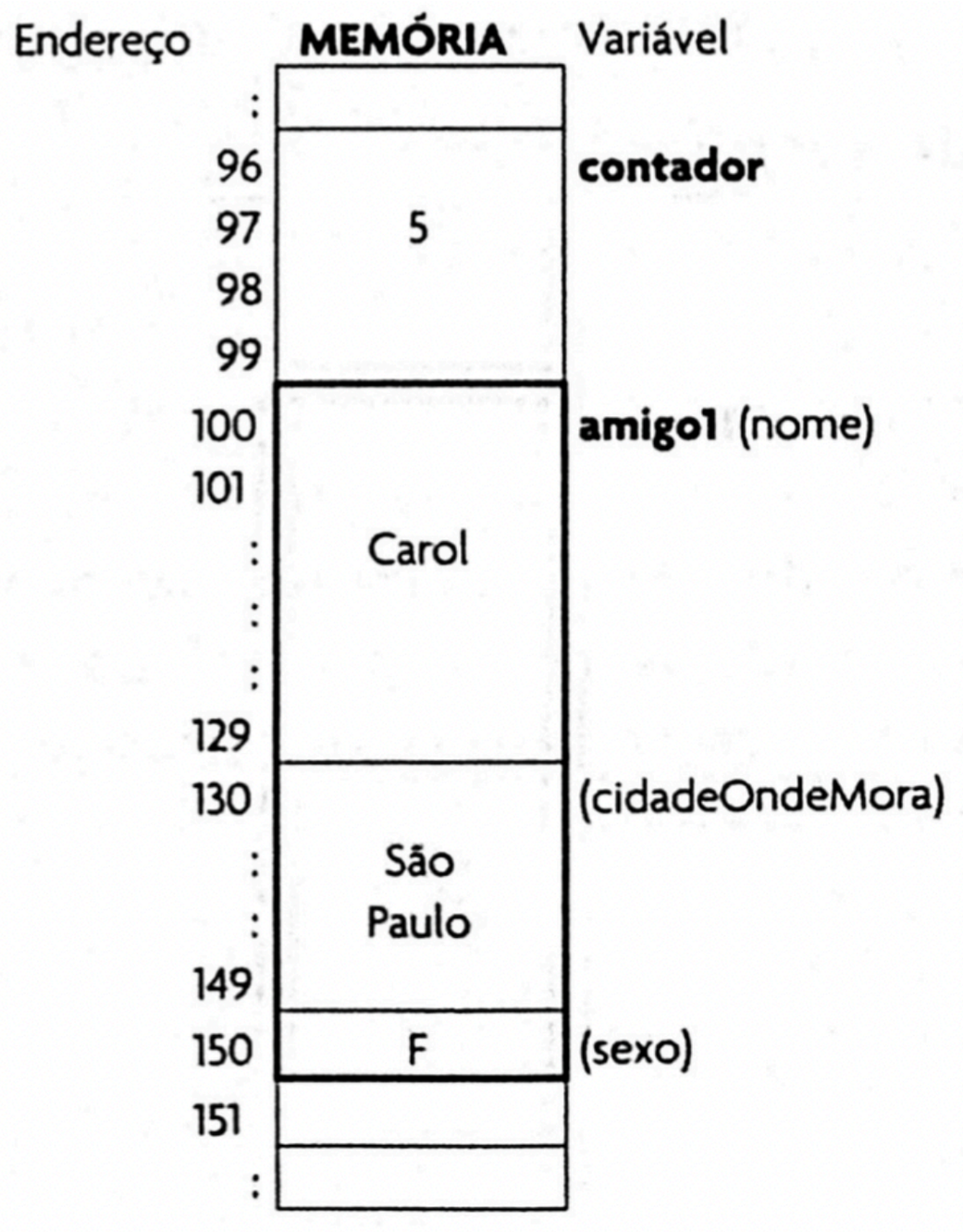
Não é mais necessário colocar struct antes do nome

Representação na Memória (cont.)

Vamos supor a declaração de duas variáveis:

- Variável int **contador**, ocupa quatro bytes. Foi atribuído o valor 5.
- Variável struct **amigo1** com três atributos:
  - char **nome[30],** ocupa 30 bytes e possui valor “Carol”
  - char **cidadeOndeMora[20]**, ocupa 20 bytes e possui valor “São Paulo”
  - char **sexo**, ocupa 1 byte e possui o valor ‘F’ (feminino)

Representação na Memória (cont.)



Representação na Memória (cont.)

- Registro **amigo1** ocupa 51 bytes (30+20+1)
  - Endereços de memória de 100 a 150
- Quando **contador** for acessada (operação de leitura ou escrita), serão acessados quatro bytes a partir do primeiro byte da variável **contador**, isto é, a partir do endereço de memória 96
- Quando **amigo1** for acessada, serão acessados 51 bytes (tamanho do registro) a partir do primeiro byte dela, isto é, a partir do endereço 100

Constantes de Enumeração

Constantes

- É tudo que é fixo, imutável …
- Pode ser um número, um caractere ou ou uma sequência qualquer de caracteres (string)
- Exemplos:
  - Numérica: 15, -15, 0.342, -2726
  - Caractere: ‘a’, ‘b’, ‘c'
  - Literal: “José da Silva”, “123456”, “*A!B?-“, “16/08/2010”

Constantes: #define

```c
#include <iostream>

#define ANO 2024
#define UNIVERSIDADE "Universidade Federal de Mato Grosso do Sul"

using namespace std;
int main () {
    printf("Ano: %d\n", ANO);
    printf("Curso: %s\n", UNIVERSIDADE);

    return 0;
}
```

Constantes: const

```c
/* Exemplo uso de const para declarar constantes */
#include <stdio.h>

using namespace std;
int main () {
    const char BIP = '\a'; // declaracao de constante
    const float PI = 3.14; // declaracao de constante
    float raio, area;

    printf("Digite o raio da esfera: ");
    scanf("%f", &raio);
    area = PI * raio * raio;

    printf("\a\a"); // BIP BIP
    printf("\nArea da esfera = %f", area);

    return 0;
}
```

Tipo Enumerados: enum

- Os tipos definidos com **enum**, chamados tipos enumerados, consistem num conjunto de constantes inteiras, em que cada uma delas é representada por um nome
- São usados quando conhecemos o conjunto de valores que uma variável pode assumir
- A variável desse tipo é sempre **int** e, para cada um dos valores do conjunto, atribuímos um nome significativo

Tipo Enumerados: enum (cont.)

- A palavra **enum** enumera a lista de nomes automaticamente, dando-lhes números em sequência (0, 1, 2, etc.)
- A vantagem é que utilizamos esses nomes no lugar de números, o que torna o programa mais claro

Exemplo 8

```
...
enum mes {JAN=1, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET, OUT, NOV, DEZ};

int main() {
    // Cria duas variáveis do tipo enum mês
    enum mes m1, m2;

    // Atribui valores
    m1 = ABR;
    m2 = JUN;

    // Operações aritméticas permitidas
    m3 = m2 - m1;

    // Comparações permitidas
    if (m1 < m2)
        ...
```

Problema: obter situação aluno

- Considere a seguinte estrutura aluno

```
struct aluno {
    float prova_1, prova_2, prova_substitutiva, exame;
    int faltas;
};
```

Exemplo 9

```
int obter_situacao_aluno (struct aluno a) {
    int flag;
    float media_aproveitamento, p1 = a.prova_1, p2 = a.prova_2;

    if(a.faltas > 18)
        flag = 0; // reprovado por falta
    else {
        if(a.prova_1 < a.prova_2 && a.prova_1 < a.prova_substitutiva)
            media_aproveitamento = (a.prova_substitutiva + a.prova_2) / 2;
        else if (a.prova_2 < a.prova_1 && a.prova_2 < a.prova_substitutiva)
            media_aproveitamento = (a.prova_1 + a.prova_substitutiva) / 2;
        else
            media_aproveitamento = (a.prova_1 + a.prova_2) / 2;

        if(media_aproveitamento >= 6.0)
            flag = 1; // aprovado
        else if (4 <= media_aproveitamento && media_aproveitamento)
            if (exame >= 6.0)
                flag = 2; // aprovado no exame
            else
                flag = 3; // reprovado no exame
        else
            flag = 4; // reprovado
    }
    return flag;
}
```

Exemplo 10

```
enum situacao_aluno {
    APROVADO, APROVADO_NO_EXAME, REPROVADO, REPROVADO_POR_FALTA, REPROVADO_NO_EXAME};
...
enum situacao_aluno obter_situacao_aluno (struct aluno a) {
    enum situacao_aluno situacao;
    float media_aproveitamento;

    if(a.n_faltas > 18)
        situacao = REPROVADO_POR_FALTA;
    else {
        if(a.prova_1 < a.prova_2 && a.prova_1 < a.prova_substitutiva)
            media_aproveitamento = (a.prova_substitutiva + a.prova_2) / 2;
        else if (a.prova_2 < a.prova_1 && a.prova_2 < a.prova_substitutiva)
            media_aproveitamento = (a.prova_1 + a.prova_substitutiva) / 2;
        else
            media_aproveitamento = (a.prova_1 + a.prova_2) / 2;

        if(media_aproveitamento >= 6.0)
            situacao = APROVADO;
        else if(media_aproveitamento >= 4.0)
            if(exame >= 6.0)
                situacao = APROVADO_NO_EXAME;
            else
                situacao = REPROVADO_NO_EXAME;
        else
            situacao = REPROVADO;
    }
    return situacao;
}
```

Referências

- ASCENCIO, A. F. G.; CAMPOS, E. A. V. Fundamentos da programação de computadores: algoritmos, Pascal, C/C ++ e Java. 3. ed. São Paulo: Pearson, 2012.
- DEITEL, P.; DEITEL, H. C: Como Programar. 6a ed. São Paulo: Pearson, 2011.
- PIVA, D.J. et al. Algoritmos e programação de computadores. Rio de Janeiro: Elsevier, 2012.