

Strings, structs e unions em C

Prof. Marcelo Veiga Neves
marcelo.neves@pucrs.br

Strings

- *Strings* são vetores unidimensionais de caracter, terminados por um caracter *null* (`\0`)
- Assim, strings contém os caracteres que compreendem a string seguidos de um *null*
- A seguinte declaração e inicialização cria uma string que consiste da palavra "Hello"

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

- O tamanho do vetor que é utilizado para armazenar os caracteres da palavra "Hello" mais um (para armazenar o *null*)

Inicialização de strings

- A declaração anterior pode ser escrita de uma forma mais concisa, que define automaticamente o tamanho do vetor e o terminador de string

```
char greeting[] = "Hello";
```

- A seguinte representação da memória será definida para a string anterior¹

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

¹Os endereços de cada caracter são fictícios

Exemplo

- Não é necessário colocar o terminador *null* no final de uma constante do tipo string, o compilador fará isso automaticamente
- O exemplo abaixo imprime uma string

```
int main ()  
{  
    char greeting[] = "Hello";  
    printf("Greeting message: %s\n", greeting );  
  
    return 0;  
}
```

Funções de biblioteca para manipulação de strings

- A biblioteca C suporta um conjunto grande de função para manipulação de strings
- Abaixo são apresentadas algumas dessas funções

Função	Propósito
<code>strcpy(s1, s2);</code>	Copia a string <i>s2</i> na string <i>s1</i> ²
<code>strcat(s1, s2);</code>	Concatena a string <i>s2</i> no final da string <i>s1</i>
<code>strlen(s1);</code>	Retorna o comprimento da string <i>s1</i>
<code>strcmp(s1, s2);</code>	Retorna 0 caso as strings sejam iguais
<code>strchr(s1, ch);</code>	Retorna a primeira ocorrência de <i>ch</i> na string <i>s1</i>
<code>strstr(s1, s2);</code>	Retorna um ponteiro para a primeira ocorrência de <i>s2</i>

²É necessário que o vetor destino possua tamanho suficiente!

Exemplo

- O exemplo abaixo utiliza algumas das funções apresentadas

```
#include <stdio.h>
#include <string.h>

int main ()
{
    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[12];
    int len ;

    strcpy(str3, str1);
    printf("strcpy(str3, str1): %s\n", str3);

    strcat(str1, str2);
    printf("strcat(str1, str2): %s\n", str1);

    len = strlen(str1);
    printf("strlen(str1): %d\n", len);

    return 0;
}
```

Structs (registros)

- Problemas reais
 - Coleções de dados de tipos diferentes
 - Exemplo: dados de um aluno
 - nota da p1
 - nota da p2
 - nota do trabalho
 - total de faltas
 - ...
- Vetores permitem definir variáveis que podem armazenar diversos itens do mesmo tipo
- *Structs* são um tipo de dados definido pelo usuário que permitem o armazenamento de itens de tipos diferentes

Struct (ou registro)

- *Struct* é um tipo de dado estruturado heterogêneo
- Coleção de variáveis referenciadas sob um mesmo nome
- Permite agrupar dados de diferentes tipos numa mesma estrutura (ao contrário de arrays, que possuem elementos de um mesmo tipo)
 - Cada componente de um registro pode ser de um tipo diferente (int, char, ...)
 - Estes componentes são referenciados por um nome
 - Os elementos do registro são chamados de *campos* ou *membros* da struct

Definindo uma struct

- Uma struct é uma definição de tipo (deve ser definida antes de ser usada, normalmente antes das funções)
- Por conveniência, pode-se utilizar em conjunto com uma substituição de tipo (*typedef*), de forma que o nome da struct possa ser usado diretamente no código³
- Abaixo é apresentada a declaração de um registro, contendo campos para notas *p1*, *p2*, *trab* e a quantidade de faltas

```
struct aluno {  
    float p1;  
    float p2;  
    float trab;  
    int faltas;  
};
```

³Não recomendado, a não ser em situações excepcionais, para evitar ofuscação do código

Definindo e utilizando variáveis do tipo criado

- Para se definir e utilizar variáveis do tipo especificado pela struct, faz-se simplesmente:

```
int main()
{
    struct aluno aluno1, aluno2;

    aluno1.p1 = 10;
    aluno1.p2 = 7.5;
    aluno1.trab = 8;
    ...
    aluno2.p1 = 6.5;
    aluno2.p2 = 4.8;
    ...
}
```

- O acesso aos campos utiliza a mesma notação empregada em Java para acessar elementos públicos de uma classe (o operador ".")

Considerações no uso

- Se o compilador C for compatível com o padrão C ANSI (praticamente todos atualmente), a informação contida em uma *struct* pode ser atribuída a outra *struct* do mesmo tipo

```
int main()
{
    struct aluno aluno1, aluno2;

    ...

    aluno2 = aluno1;
}
```

- No exemplo, todos os campos de *aluno2* receberão os valores correspondentes aos campos de *aluno1*

Passagem por valor com structs

- Structs podem ser passadas por valor ou por referência
- No exemplo abaixo, um função recebe uma struct representando uma fração, calcula e retorna o valor

```
struct fracao_s {  
    float numerador;  
    float denominador;  
};  
  
float calcula(struct fracao_s frac)  
{  
    return frac.numerador / frac.denominador;  
}  
  
int main()  
{  
    struct fracao_s f1;  
  
    f1.numerador = 10.5;  
    f1.denominador = 2.75;  
  
    printf("Valor: %f\n", calcula(f1));  
}
```

Passagem por referência com structs

- É recomendado que sempre que possível, passar por referência, pois o custo é menor (passa apenas um ponteiro ao invés de copiar a estrutura inteira)
- Nesse caso, como é enviado um ponteiro, o acesso deve ser feito da seguinte forma:

```
float calcula(struct fracao_s *frac)
{
    return (*frac).numerador / (*frac).denominador;
}
```

- Uma forma mais curta e direta é através do operador seta (\rightarrow)⁴

```
float calcula(struct fracao_s *frac)
{
    return frac->numerador / frac->denominador;
}
```

⁴Lembre de passar a referência à struct (operador $\&$) para função *calcula*

Vetores de struct

- É possível realizar a criação de vetores (de qualquer dimensão) de tipos estruturados
- O exemplo abaixo ilustra tal uso, definindo registros de uma turma de alunos

```
struct aluno_s {  
    float p1;  
    float p2;  
    float trab;  
    int faltas;  
};  
  
int main()  
{  
    struct aluno_s alunos[50];  
  
    alunos[0].p1 = 10.0;  
    alunos[0].p2 = 7.5;  
    alunos[0].trab = 8.0;  
    ...  
    alunos[1].p1 = 6.5;  
    alunos[1].p2 = 4.8;  
    ...  
}
```

Unions

- *Union* é um tipo especial de dado que permite armazenar diferentes tipos na mesma região de memória
- Pode-se definir uma *union* com diversos membros, mas apenas um membro pode conter um valor por vez
- Unions permitem uma maneira eficiente de utilizar a mesma região de memória para múltiplos propósitos
- Uma union pode ser definida da mesma forma que uma struct

```
union data_u {  
    int i;  
    float f;  
    char str[20];  
};
```

- No exemplo, uma variável do tipo *union data_u* pode armazenar um inteiro, um número real ou uma string
- A memória ocupada pela union será o suficiente para armazenar o maior membro

Exemplo

- O exemplo abaixo ilustra o uso de union

```
union data_s {
    int i;
    float f;
    char str[20];
};

int main()
{
    union data_s data;

    printf("Uso de memória: %ld\n", sizeof(data));
    data.i = 1234;
    data.f = 643.1234; // o valor de data.i é sobrescrito
    printf("Valor correto: %f, valor errado: %d\n",
        data.f, data.i);
    strcpy(data.str, "oi, tudo bem?");
    printf("String: %s\n", data.str);

    return 0;
}
```


Referências

- Este material foi construído pelo Prof. Sérgio Johann Filho com base nos slides de aula do prof. Marcelo Cohen