Programação 2

Aula 3

Thiago Cavalcante – thiago.cavalcante@penedo.ufal.br

06 de novembro de 2019

Universidade Federal de Alagoas – UFAL
Campus Arapiraca
Unidade de Ensino de Penedo

Funções

Bloco de código que pode ser **nomeado** e **chamado** dentro de um programa

Exemplos: `scanf()` e `printf()`

Por que usar funções?

- Estruturação do programa
- Reutilização de código

Declarando uma função

```
tipo_retornado nome_função(parâmetros) {
    declarações e comandos
}
```

Nome da função segue regras de nome de variáveis

Local da declaração

1. Declaração antes da **`main()`**
2. Protótipo antes da **`main()`** e declaração depois

O protótipo contém apenas o **cabeçalho** da função

Funcionamento de uma função

- Fluxo do programa é interrompido
- Valores são copiados para parâmetros
- Comandos da função são executados
- Valor do **`return`** é copiado para variável
- Fluxo do programa continua

Lista de parâmetros

```
(tipo1 nome1, tipo2 nome2, ..., tipoN nomeN)
```

o tipo deve vir antes de **cada** parâmetro, mesmo se os tipos forem iguais (ex.: `int x, int y`)

o parâmetro só pode ser acessado **dentro** da função

A lista não é obrigatória

```
tipo nome() {
  ...
}
```

```
tipo nome(void) {
  ...
}
```

Existe uma diferença entre as declarações

Corpo da função

- Sequência de declarações
- Sequência de comandos

A **`main()`** é uma função presente cada programa

Tudo o que é feito na **`main()`** pode ser feito em outras funções

Em geral, **evita-se fazer leitura e escrita** de dados dentro de uma função

Uma possível exceção: **menu de usuário**

Retorno da função

```
return expressão;
```

Uma função pode retornar **qualquer um** dos tipos válidos em C (incluindo tipos definidos pelo usuário)

Retorno da função

Uma função pode **não retornar** nada também, basta definir o `tipo_retornado` como **`void`**

Exemplo: uma função para imprimir algo na tela

Uma função pode retornar:

- Variável
- Constante
- Expressão artimética
- Expressão lógica
- Outra função

O retorno precisa ser compatível com o tipo definido

Uma função pode ter **vários** comandos `return`

```
int max(int x, int y) {
  if(x > y)
    return x;
  else
    return y;
}
```

```
int max(int x, int y) {
  int z;
  if(x > y)
    z = x;
  else
    z = y;
  return z;
}
```

Funções

- A função **encerra** quando chega em um **`return`**
- Uma função do tipo **`void`** pode ser finalizada com **`return;`**
- Uma função **não pode** retornar um **array** (a não ser que esteja dentro de uma `struct`)

Passagem de parâmetros

- Por **valor**
- Por **referência**

Passagem por **valor**

- O argumento é **copiado** para o parâmetro
- O parâmetro é uma **variável local** da função
- Mudanças no parâmetro **não refletem** no argumento
- O parâmetro é **destruído** e o argumento mantém seu valor **original**

Passagem por **referência**

- Usada quando se quer **alterar** o valor do argumento
- Não é passado o valor da variável, mas sim o seu **endereço na memória**
- Na **declaração** e **corpo** da função, usa-se o operador `*` antes da variável
- Na **chamada** da função, usa-se o operador **&** antes da variável

Exemplo: `scanf("%d", &x);`

Passagem de arrays como parâmetros

- Arrays são **sempre** passados **por referência**
- É necessário sempre um segundo parâmetro contendo o **tamanho** do array

Declarando os parâmetros

- `int *array, int tamanho`
- `int array[], int tamanho`
- `int array[5], int tamanho`

Os primeiros elementos são todos equivalentes, pois o tamanho não é checado

Não é necessário usar o operador & na **chamada** da função para um array, pois o seu **nome representa o endereço do seu primeiro elemento** na memória

```
array = &array[0]
```

Não é necessário usar o operador `*` no **corpo** da função para um array. O **acesso aos seus elementos pode ser feito normalmente** com colchetes.

```
x = array[elemento];
```

Arrays multidimensionais

É necessário indicar o tamanho das dimensões extras na declaração da função.

```
int array[][5], int tamanho
```

Passagem de structs como parâmetros

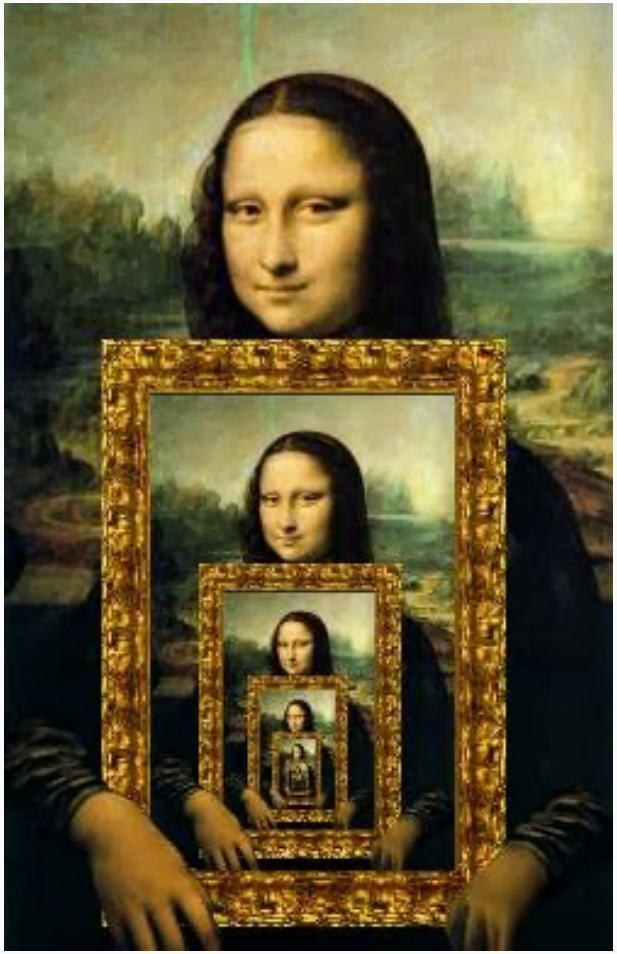
- Passagem por **valor**
  - estrutura
  - campo da estrutura
- Passagem por **referência**
  - estrutura
  - campo da estrutura

Operador seta

Quando a estrutura é passada por referência, o acesso a ela dentro da função pode ser feito com os operadores `*` e `.` em conjunto ou com o operador `->`

```
(*struct).campo
struct->campo
```

Recursão (definição circular)



Uma *funçao recursiva* **chama a si própria** dentro da sua definição.

Exemplo clássico: **fatorial**

Como funciona a recursividade

- Dividir e conquistar
- Caminho de ida $\rightarrow$ caso-base $\rightarrow$ caminho de volta

Cuidados a serem tomados

- Critério de parada
- Parâmetro da chamada recursiva

Algoritmos recursivos são **considerados mais enxutos/elegantes**, porém **tendem a ser mais ineficientes** (tempo/memória) e apresentam maior dificuldade na detecção de erros

Ex.: **Fibonacci**