UNIPAC

Universidade Presidente Antônio Carlos

Bacharelado em Ciência da Computação

Introdução a Programação

Material de Apoio

Parte IX - Funções

Prof. Nairon Neri Silva

naironsilva@unipac.br

1º sem / 2020

Material cedido pela professora Lívia

Funções

- Funções são as estruturas que permitem ao usuário separar seus programas em blocos. Se não as tivéssemos, os programas teriam que ser curtos e de pequena complexidade. Para fazermos programas grandes e complexos temos de construí-los bloco a bloco.
- Sintaxe:

```
<tipo_de_retorno> <nome_da_função> (<declaração_de_parâmetros>)
{
    <corpo_da_função>
}
```

Funções

- O *tipo-de-retorno* é o tipo de variável que a função vai retornar. O default é o tipo `int`, ou seja, uma função para qual não declaramos o tipo de retorno é considerada como retornando um inteiro.

Funções

- A declaração de parâmetros é uma lista com a seguinte forma geral:

```
tipo nome1, tipo nome2, ... , tipo nomeN
```

- Repare que o tipo deve ser especificado para cada uma das N variáveis de entrada. É na declaração de parâmetros que informamos ao compilador quais serão as entradas da função (assim como informamos a saída no tipo-de-retorno).

Funções

- O corpo da função é a sua alma. É nele que as entradas são processadas, saídas são geradas ou outras coisas são feitas.

- O comando *return* tem a seguinte forma geral:

```
        return valor_de_retorno;
ou      return;
```

Funções

Digamos que uma função está sendo executada.

Quando se chega a uma declaração return a função é encerrada imediatamente e, se o valor de retorno é informado, a função retorna este valor.

É importante lembrar que o valor de retorno fornecido tem que ser, pelo menos, compatível com o tipo de retorno declarado para a função.

Funções

- Uma função pode ter mais de uma declaração return. Isto se torna claro quando pensamos que a função é terminada quando o programa chega à primeira declaração return.

- É importante notar que, como as funções retornam valores, podemos aproveitá-los para fazer atribuições. Mas não podemos fazer:

```
func(a,b)=x; /* Errado! */
```

Funções - Exemplo

```
#include <stdio.h>
int EPar (int a){
  if (a%2!=0) /* Verifica se a e divisivel por dois */
    return 0; /* Retorna 0 se nao for divisivel */
  else
    return 1; /* Retorna 1 se for divisivel */
}

void main (){
  int num;
  printf ("Entre com numero: ");
  scanf ("%d",&num);
  if (EPar(num)==1)
     printf ("\n\nO numero e par.\n");
  else
     printf ("\n\nO numero e impar.\n");
}
```

Funções - Protótipo

- Até agora, nos exemplos apresentados, escrevemos as funções antes de escrevermos a função main(). Isto é, as funções estão fisicamente antes da função main(). Isto foi feito por uma razão.

Funções - Protótipo

- Imagine-se na pele do compilador. Se você fosse compilar a função main(), onde são chamadas as funções, você teria que saber com antecedência quais são os tipos de retorno e quais são os parâmetros das funções para que você pudesse gerar o código corretamente. Foi por isto que as funções foram colocadas antes da função main(): quando o compilador chegasse à função main() ele já teria compilado as funções e já saberia seus formatos.

Funções - Protótipo

- Mas, muitas vezes, não poderemos nos dar ao luxo de escrever nesta ordem. Muitas vezes teremos o nosso programa espalhado por vários arquivos. Ou seja, estaremos chamando funções em um arquivo que serão compiladas em outro arquivo. Como manter a coerência?

Funções - Protótipo

- A solução são os protótipos de funções. Protótipos são nada mais, nada menos, que declarações de funções. Isto é, você declara uma função que irá usar. O compilador toma então conhecimento do formato daquela função antes de compilá-la. O código correto será então gerado.

- Um protótipo tem o seguinte formato:

```
<tipo_de_retorno> <nome_da_função> (declaração_de_parâmetros);
```

Funções – Tipo *void*

- Em inglês, *void* quer dizer vazio e é isto mesmo que o *void* é. Ele nos permite fazer funções que não retornam nada e funções que não têm parâmetros!

Funções – Tipo *void*

- Exemplo:

```
void <nome_da_função> (declaração_de_parâmetros);
```

- Numa função, como a acima, não temos valor de retorno na declaração return. Aliás, neste caso, o comando return não é necessário na função.
- Podemos, também, fazer funções que não têm parâmetros:

```
<tipo_de_retorno> <nome_da_função> (void);
```

Exercícios não avaliativos

1) Crie uma função que receba um valor e informe se ele é positivo ou não.

2) Crie uma função em linguagem C que receba 2 números e retorne o maior valor.

3) Crie uma função em linguagem C que receba 3 números e retorne o maior valor, use a função da questão anterior.

Não deixe de fazer esses exercícios. Não precisa enviar o código, somente resolver para compreender o conteúdo até aqui.

Passagem de Parâmetros para Funções

- Já vimos que, na linguagem C, quando chamamos uma função os parâmetros formais da função copiam os valores dos parâmetros que são passados para a função. Isto quer dizer que não são alterados os valores que os parâmetros têm fora da função. Este tipo de chamada de função é denominado chamada por valor. Isto ocorre porque são passados para a função apenas os valores dos parâmetros e não os próprios parâmetros.

Passagem de Parâmetros para Funções

- Outro tipo de passagem de parâmetros para uma função ocorre quando alterações nos parâmetros formais, dentro da função, alteram os valores dos parâmetros que foram passados para a função.

Passagem de Parâmetros para Funções

- Este tipo de chamada de função tem o nome de "chamada por referência". Este nome vem do fato de que, neste tipo de chamada, não se passa para a função os valores das variáveis, mas sim suas referências (a função usa as referências para alterar os valores das variáveis fora da função).
- A linguagem C só permite passagem de parâmetros por valor. Há entretanto um recurso de programação que podemos usar para simular uma chamada por referência.

Passagem de parâmetros para Funções

- Quando queremos alterar as variáveis que são passadas para uma função, nós podemos declarar seus parâmetros formais como sendo ponteiros. Os ponteiros são a "referência" que precisamos para poder alterar a variável fora da função. O único inconveniente é que, quando usarmos a função, teremos de lembrar de colocar um & na frente das variáveis que estivermos passando para a função.

Exemplo

```
#include <stdio.h>

void funcao(int *n){
    *n = 20;
}

int main(){

    int a;
    a = 10;
    printf("%d \n", a);
    funcao(&a);
    printf("%d", a);
    return 0;
}
```

Vetores como argumento de funções

- Quando vamos passar um vetor como argumento de uma função, podemos declarar a função de três maneiras equivalentes. Digamos que temos a seguinte:

```
int vet [50];
```

- E que queiramos passá-los como argumento de uma função func(). Podemos declarar func() das três maneiras seguintes:

```
void func (int vet[50]);
void func (int vet[]);
void func (int *vet);
```

Vetores como argumento de funções

- Nos três casos, teremos dentro de func() um int* chamado vet. Ao passarmos um vetor para uma função, na realidade estamos passando um ponteiro. Neste ponteiro é armazenado o endereço de memória do primeiro elemento do vetor. Isto significa que não é feita uma cópia, elemento a elemento do vetor. Isto faz com que possamos alterar o valor desse vetor dentro da função.

Matriz como argumento de funções

- Quando vamos passar uma matriz como argumento de uma função, podemos declarar a função de três maneiras equivalentes. Digamos que temos a seguinte:

```
int m[50][50];
```

- E que queiramos passá-los como argumento de uma função func(). Podemos declarar func() das três maneiras seguintes:

```
void func (int m[50][50]);
void func (int m[][50]);
void func (int *m[][50]);
```

Exercícios não avaliativos

1) Crie uma função que receba um vetor de tamanho 30 (gerado randomicamente) com números de 1 a 100 e um valor inteiro. Retorne quantas vezes o número inteiro está presente no vetor. Exiba o retorno no método main().

2) Crie uma função que receba um vetor de 10 elementos (gerados randomicamente) com números de 1 a 10 e um valor inteiro. Altere todos os elementos do vetor multiplicando o valor da posição pelo número informado. Imprima o vetor alterado no método main().