Prof. MSc. Raphael Gomes (raphael@ifg.edu.br)

Módulos em C – Parte II

Nesta apostila continuaremos a estudar funções em C.

2.1 – Sobrecarga de funções

A linguagem C oferece uma ferramenta poderosa – sobrecarga de funções. É possível sobrescrever uma função de tal forma que um mesmo nome é compartilhado por mais de uma função, mas cada função executa uma ação diferente. Este recurso é essencial quando é necessário manter o mesmo nome para diferentes tipos de argumentos. Há diversos casos onde isso é necessário. Por exemplo, se você precisa executar uma operação em uma entrada mas essa entrada pode ser tanto um número inteiro quanto real. Você pode sobrecarregar a função em questão e ter uma versão que recebe um inteiro como argumento e outra que recebe um número real como argumento. Veja abaixo um exemplo:

```
#include <stdio.h>

// protótipos das funções
float cube_number(float num);
int cube_number(int num);

int main() {
	float number;
	float number4;

	printf("Por favor forneça um número: \n");
	scanf(“%f”, &number);

	number4 = cube_number(number);

	printf(“%f ao cubo é %f \n", number, number4);

	return 0;
}

int cube_number(int num) {
	int answer;
	answer = num * num * num;
	return answer;
}

float cube_number(float num) {
	float answer;
	answer = num * num * num;
	return answer;
}
```

Veja que há duas funções chamadas cube_number. Ambas recebem um único parâmetro e retorna este parâmetro ao cubo. Contudo, a primeira função recebe um inteiro e retorna um inteiro como resposta, enquanto a segunda função recebe um número real e retorna um número real como resposta. A grande pergunta é: como o compilador sabe qual função foi chamada? A resposta é simples, ele sabe qual função você chama baseado no tipo de argumento que está sendo passado. Se você passa um argumento inteiro então

a função chamada é aquela que recebe um número inteiro. Por causa disso, quando se usa sobrecarga de função elas devem ter tipos ou números diferentes de parâmetros.

2.2 – Arquivos header

Até agora nós apenas usamos arquivos header que fazem parte da biblioteca padrão de C. Arquivos header são uma forma de organizar o código de um programa em C e, além disso, permitir o reaproveitamento de código em programas mais extensos. Estes arquivos contém as definições e os protótipos das funções de um programa. Para criar um arquivo header basta salvá-lo com a extensão .h.

O exemplo a seguir ilustra o uso de arquivos header:

Etapa 1: Abra um editor de texto e digite o seguinte código:

```
	// protótipos das funções
	float cube_number(float num);
	int cube_number(int num);
```

Salve este arquivo como test.h.

Etapa 2: Crie um novo arquivo com o código abaixo:

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

int main() {
	float number;
	float number4;

	printf("Por favor forneça um número: \n");
	scanf(“%f”, &number);

	number4 = cube_number(number);

	printf(“%f ao cubo é %f \n", number, number4);

	return 0;
}

int cube_number(int num) {
	int answer;
	answer = num * num * num;
	return answer;
}

float cube_number(float num) {
	float answer;
	answer = num * num * num;
	return answer;
}
```

O exemplo acima foi usando apenas para introduzir o conceito de arquivos header. Contudo, em um programa mais complexo é necessário separar as partes que o compõe em diferentes arquivos para facilitar a manutenção. Certas partes deste código certamente são usadas com muita frequência. Consequentemente, o arquivo header referente a estas partes deverá ser incluído mais de uma vez no programa. Contudo, o compilador irá isso interpretar como conflito uma vez que as funções estão sendo definidas mais de uma vez (pois os protótipos estão sendo incluídos mais de uma vez).

Nesse cenário devemos usar uma diretiva para informar ao compilador que, se o código já tiver sido incluído anteriormente, o #include simplesmente será ignorado. A alteração se dará apenas no arquivo test.h conforme abaixo:

```
#ifndef TEST_H
#define     TEST_H
      // protótipos das funções
      float cube_number(float num);
      int cube_number(int num);
#endif
```

Neste caso, TEST_H é apenas um nome usado para identificar o arquivo header, podendo ser substituído por qualquer informação. Contudo, o padrão é usar o próprio nome do arquivo como fizemos no exemplo.

2.3 – Passando valores por referência

Quando você passa um valor para uma função, você na verdade copia o conteúdo de uma variável para outra variável – a outra variável é o parâmetro da função. Isso significa que o parâmetro é, essencialmente, uma cópia da variável que é passada para a função. Isso significa que se você alterar o valor do parâmetro você não altera o valor da variável original. Vejamos um exemplo:

```
#include <stdio.h>

void demo(float number);

int main () {
      float num1;
      printf("Por favor digite um número: \n");
      scanf(“%f”, &num1);
      printf("Antes da função demo o número é %f \n”, num1);
      demo(num1);
      printf("Após a função demo o número é %f \n”, num1);
       return 0;
}

void demo(float number) {
      number = number * 4;
      printf("Dentro da função demo o número é %f \n”, number);
}
```

Como você pode ver através deste exemplo, o que ocorre dentro da função não interfere nos valores das variáveis fora da função. Quando você chama a função demo, você pega o valor que está em num1 e coloca este valor no parâmetro da função number. Number é uma variável completamente nova que ocupa seu próprio espaço de memória, separado de num1. Isto é chamado **passagem por valor**.

Há outra maneira de passar um parâmetro: **por referência**. Quando você passa por referência você não só copia o valor da variável para o parâmetro. O que você passa é uma referência para o endereço da variável na memória. Isto significa que o parâmetro da função não é uma nova variável. Ele é simplesmente um novo nome para a variável. Tanto o nome da variável quanto o nome do parâmetro se referem ao mesmo espaço na memória. Isso é possível colocando & da definição do parâmetro. Vejamos o exemplo anterior usando passagem por referência.

```
#include <stdio.h>

void demo(float &number);

int main () {
      float num1;
      printf("Por favor digite um número: \n");
      scanf(“%f”, &num1);
      printf("Antes da função demo o número é %f \n”, num1);
      demo(num1);
      printf("Após a função demo o número é %f \n”, num1);
       return 0;
}
```

```
void demo(float &number) {
      number = number * 4;
      printf("Dentro da função demo o número é %f \n”, number);
}
```

Veja que agora quando o valor é mudado dentro da função ele também é alterado na variável fora da função. Isso acontece porque na verdade foi passado o endereço da variável para a função. Tudo que é feito usando o apelido criado se reflete na variável original.