6. Funções

Funções (também chamadas de **rotinas**, ou **sub-programas**) são a essência da programação estruturada. Funções são segmentos de programa que executam uma determinada tarefa específica. Já vimos o uso de funções nos capítulos anteriores: funções já providenciadas pelas bibliotecas-padrão do C (como `sqrt()`, `toupper()`, `getch()` ou `putchar()`).

É possível ao programador, alem disso, escrever suas próprias rotinas. São as chamadas de **funções de usuário** ou rotinas de usuário. Deste modo pode-se segmentar um programa grande em vários programas menores. Esta segmentação é chamada de **modularização** e permite que cada segmento seja escrito, testado e revisado individualmente sem alterar o funcionamento do programa como um todo. Permite ainda que um programa seja escrito por vários programadores ao mesmo tempo, cada um escrevendo um segmento separado. Neste capítulo, veremos como escrever funções de usuário em C.

6.1 Estrutura das funções de usuário

A estrutura de uma função de usuário é muito semelhante a estrutura dos programas que escrevemos até agora. Uma função de usuário constitui-se de um **bloco de instruções** que definem os procedimentos efetuados pela função, um **nome** pelo qual a chamamos e uma **lista de argumentos** passados a função. Chamamos este conjunto de elementos de **definição da função**.

Exemplo: o código mostrado abaixo é uma função definida pelo usuário para calcular a média aritmética de dois números reais:

```
float media2(float a, float b){
  float med;
  med = (a + b) / 2.0;
  return(med);
}
```

No exemplo acima definimos uma função chamada `media2` que recebe dois argumentos tipo `float`: `a` e `b`. A média destes dois valores é calculada e armazenada na variável `med` declarada internamente. A função retorna, para o programa que a chamou, um valor também do tipo `float`: o valor da variável `med`. Este retorno de valor é feito pela função `return()`que termina a execução da

função e retorna o valor de `med` para o programa que a chamou.

Depois de definimos um função, podemos usá-la dentro de um programa qualquer. Dizemos que estamos fazendo uma **chamada** a função.

Exemplo: No exemplo abaixo chamamos a função `media2()` dentro de um programa principal:

```
void main(){
  float num_1, num_2, med;
  puts("Digite dois números:");
  scanf("%f %f", &num_1, &num_2);
   med = media2(num_1, num_2);   // chamada a função
  printf("\nA media destes números é %f", med);
}
```

6.2 Definição de funções

De modo formal, a sintaxe de uma função é a seguinte:

```
tipo_de_retorno nome_da_função(tipo_1 arg_1, tipo_2 arg_2, ...){
  [bloco de instruções da função]
}
```

A primeira linha da função contém a **declaração** da função. Na declaração de uma função se define o **nome** da função, seu **tipo de retorno** e a **lista de argumentos** que recebe. Em seguida, dentro de chaves `{}`, definimos o bloco de instruções da função.

O **tipo de retorno** da função especifica qual o tipo de dado retornado pela função, podendo ser qualquer tipo de dado mostrado na seção 2.3: `int`, `float`, etc. Se a função não retorna nenhum valor para o programa que a chamou devemos definir o retorno como `void`, ou seja um retorno ausente. Se nenhum tipo de retorno for especificado o compilador entenderá que o retorno será tipo `int`.

Vale notar que existe apenas **um** valor de retorno para funções em C. **Não** podemos fazer o retorno de **dois ou mais** valores como em algumas linguagens (no MatLab: `[media,desvio] = estat(a, b, c, d, e)`). Porém isto não é um limitação séria pois o uso de ponteiros (cap. ?) contorna o problema.

Por ser um identificador, o **nome da função** segue as mesmas regras de definição de identificadores (veja seção 2.2).

A **lista de argumentos** da função especifica quais são os valores que a função recebe. As variáveis da lista de argumentos são manipuladas normalmente no corpo da função (veja seção 6.5 adiante).

A chamada de uma função termina com a instrução `return()` que transfere o controle para o programa chamador da função. Esta instrução tem duas finalidades: determina o **fim lógico** da rotina e o **valor de retorno** da função. O argumento de `return()` será retornado como valor da função.

6.3 Localização das funções

Existem basicamente duas posições possíveis para escrevermos o corpo de uma função: ou **antes** ou **depois** do programa principal. Podemos ainda escrever uma função no **mesmo arquivo** do programa principal ou em **arquivo separado**.

6.3.1 Corpo da função *antes* do programa principal (no mesmo arquivo)

Quando escrevemos a definição de uma função **antes** do programa principal e no mesmo arquivo deste, nenhuma outra instrução é necessária. A sintaxe geral para isto é a seguinte:

Sintaxe: Uma função escrita antes do programa principal:

```
tipo nomef(...){            // definição da função
  [corpo de função]
}
void main(){                // programa principal
  ...
  var = nomef(...)          // chamada da função
  ...
}
```

Exemplo: Função `media2()` escrita antes do programa principal.

```
float media2(float a, float b){  // função
```

```
  float med;
  med = (a + b) / 2.0;
  return(med);
}
void main(){                    // programa principal
  float num_1, num_2, med;
  puts("Digite dois números:");
  scanf("%f %f", &num_1, &num_2);
   med = media2(num_1, num_2);    // chamada da função
  printf("\nA media destes números e´ %f", med);
}
```

Programa exemplo: No arquivo `e0601.cpp` existe um programa que calcula o maior valor entre dois números digitados. Este programa faz uso da função `max()`escrita pelo usuário.

6.3.2 Corpo da função *depois* do programa principal (no mesmo arquivo)

Quando escrevemos a definição de uma função **depois** do programa principal e no mesmo arquivo deste, devemos incluir um **protótipo** da função chamada. Um protótipo é uma instrução que define o nome da função, seu tipo de retorno e a quantidade e o tipo dos argumentos da função. O protótipo de uma função indica ao compilador quais são as funções usadas no programa principal os tipo. A sintaxe geral para isto é a seguinte:

Sintaxe: Uma função escrita depois do programa principal:

```
void main(){                // programa principal
  tipo nomef(...);          // protótipo da função
  ...
  var = nomef(...)          // chamada a função
  ...
}
tipo nomef(...){            // definição da função
  [corpo de função]
}
```

Exemplo: Função `media2()` escrita depois do programa principal.

```
void main(){                    // programa principal
  float media2(float,float);    // protótipo de media2()
```

```
  float num_1, num_2, med;
  puts(”Digite dois números:”);
  scanf(”%f %f”, &num_1, &num_2);
   med = media2(num_1, num_2);    // chamada a função
  printf(”\nA media destes números e´ %f”, med);
}
float media2(float a, float b){  // função media2()
  float med;
  med = (a + b) / 2.0;
  return(med);
}
```

Observe que o protótipo de uma função nada mais é que a **declaração** da função sem o seu corpo. Observe ainda que na lista de argumentos do protótipo podem ser escritos apenas os **tipos** dos argumentos.

Programa exemplo: No arquivo `e0602.cpp` existe um programa que calcula o maior valor entre dois números digitados. Este programa faz uso da função `max()`escrita pelo usuário.

6.3.3 Corpo da função escrito em *arquivo separado*

Em C, como em muitas outras linguagens, é permitido que o usuário crie uma função em **um arquivo** e um programa que a chame em outro **arquivo distinto**. Esta facilidade permite a criação de **bibliotecas de usuário**: um conjunto de arquivos contendo funções escritas pelo usuário. Esta possibilidade é uma grande vantagem utilizada em larga escala por programadores profissionais.

Quando escrevemos a definição de uma função em **arquivo separado** do programa principal devemos **incluir** este arquivo no conjunto de arquivos de **compilação** do programa principal. Esta inclusão é feita com a diretiva `#include`. Esta diretiva, vista nas seções 2.4.2 e 3.7.1, instrui o compilador para incluir na compilação do programa outros arquivos que contem a definição das funções de usuário e de biblioteca.

Sintaxe: A sintaxe de inclusão de funções de usuário é a seguinte:

```
#include ”pathH”          // inclusão da função
void main(){              // programa principal
...
  var = nomef(...)        // chamada a função
```

```
...
}
```

Na diretiva `#include`, indicamos entre aspas duplas o caminho de localização do arquivo onde está definida a função chamada.

Exemplo: A função `media2()` está escrita em um arquivo separado.

```
#include ″c:\tc\userbib\stat.h″     // inclusão da função
void main(){                        // programa principal
  float num_1, num_2, med;
  puts(″Digite dois números:″);
  scanf(″%f %f″, &num_1, &num_2);
   med = media2(num_1, num_2);    // chamada a função
  printf(″\nA media destes números é %f″, med);
}
```

Programa exemplo: No arquivo `e0603.cpp` existe um programa que calcula o maior valor entre dois números digitados. Este programa faz uso da função `max()`escrita pelo usuário no arquivo `e0604.cpp`.

Observação: Um arquivo pode conter a definição de uma ou mais funções. Em geral, quando o arquivo possui apenas uma função ele é nomeado com o mesmo nome da função e extensão `*.cpp` ou `*.c`. Por exemplo, poderíamos definir a função `media()` no arquivo `media.cpp`. Quando um arquivo possui a definição de mais de uma função, ele é nomeado com a extensão `*.h` ou `*.lib`. Por exemplo: poderíamos criar um conjunto de funções estatísticas chamadas `media(), dsvpd(), moda(), max(), min(),` etc. definindo-as em um arquivo chamado `stat.h`.

6.4 Hierarquia de Funções

Sempre é possível que um programa principal chame uma função que por sua vez chame outra função... e assim sucessivamente. Quando isto acontece dizemos que a função chamadora tem hierarquia maior (ou superior) a função chamada. Ou que a função chamadora está em um nível hierárquico superior a função chamada.

Quando isto ocorre, devemos ter o cuidado de definir (ou incluir) as funções em ordem crescente de hierarquia, isto é, uma função **chamada** é escrita antes de uma função **chamadora**. Isto se deve ao fato de que o compilador deve conhecer uma função antes de que chamada seja compilada.

Programa exemplo: No arquivo `e0605.cpp` existe um jogo de "*jackpot*" que ilustra o uso de várias rotinas por um programa principal. Observe também que estas funções chamam-se umas as outras.

Neste programa exemplo, os níveis hierárquicos das funções podem ser colocados da seguinte maneira:

```
                                main()

  regras()  abertura()  sorte()  plim_plim()  saida()  simnao()

                          imprimec()   roleta()

                             pinta()   bip()
```

No exemplo acima temos um primeiro nível onde se encontra a função main() [o programa principal também é uma função] que chama as funções x, y, z. A função x por sua vez chamas as funções s, r, e t. Observe que neste exemplo os protótipos das funções estão colocados de modo a que as funções de menor hierarquia são escritas antes das funções de maior hierarquia.

6.5 Regra de escopo para variáveis

A regra de escopo define o **âmbito de validade** de variáveis. Em outras palavras define onde as variáveis e funções são reconhecidas. Em C, uma variável só pode ser **usada** após ser **declarada** (ver seção 2.3.2). Isto por que o processador deve reservar um local da memória para armazenar os valores atribuídos à variável. Porém o local, do programa, onde uma variável é declarada define ainda seu **escopo** de validade. Uma variável pode ser **local**, **global** ou **formal** de acordo com o local de declaração.

Variáveis Locais: Uma variável é dita *local*, se for declarada **dentro do bloco** de uma função. Uma variável local tem validade apenas dentro do bloco onde é declarada, isto significa que podem ser apenas acessadas e modificadas dentro de um bloco. O espaço de memória alocado para esta variável é **criado** quando a execução do bloco é iniciada e **destruído** quando encerrado, assim variáveis de mesmo nome mas declaradas em blocos distintos, são para todos os efeitos, variáveis distintas.

Exemplo:

```
float media2(float a, float b){
  float med;
  med = (a + b) / 2.0;
  return(med);
}
void main(){
  float num_1, num_2, med;
  puts(”Digite dois números:”);
  scanf(”%f %f”, &num_1, &num_2);
   med = media2(num_1, num_2);
  printf(”\nA media destes números e´ %f”, med);
}
```

No exemplo acima, `med` é uma variável local definida pela função `media()`. Outra variável `med` é também definida pela função `main()`. Para todos os efeitos estas variáveis são distintas.

Variáveis Formais: Uma *variável formal* é uma variável local declarada na **lista de parâmetros** de uma função. Deste modo, tem validade apenas dentro da função onde é declarada, porém serve de suporte para os valores passados pelas funções. As variaveis formais na **declaração** da função e na **chamada** da função podem ter nomes distintos. A única exigência é de que sejam do mesmo tipo.

Por serem variáveis locais, os valores que uma função passa para outra não são alterados pela função chamada. Este tipo de passagem de argumentos é chamado de **passagem por valor** pois os valores das variáveis do programa chamador são copiados para as correspondentes variáveis da função chamada. Veremos no capitulo ? como alterar os valores das variáveis do programa chamador. Chamaremos esta passagem de **passagem por endereço**.

No exemplo acima, `a` e `b` são parâmetros formais declarados na função media2(). Observe que a função é chamada com os valores de `num_1` e `num_2`. Mesmo que os valores de `a` e `b` fossem alterados os valores de `num_1` e `num_2` não seriam alterados.

Variáveis Globais: Uma variável é dita *global*, se for declarada **fora do bloco** de uma função. Uma variável global tem validade no escopo de todas as funções, isto é, pode ser acessadas e modificada por qualquer função. O espaço de memória alocado para esta variável é **criado** no momento de sua declaração e **destruído** apenas quando o programa é encerrado.

Exemplo: Uso de variáveis globais.

```
float a, b, med;
void media2(void){
med = (a + b) / 2.0;
}
void main(){
  puts(”Digite dois números:”);
  scanf(”%f %f”, &a, &b);
 media2();
  printf(”\nA media destes números e´ %f”, med);
}
```

No exemplo acima, `a, b, med` são variáveis globais definidas fora dos blocos das funções `media()` e `main()`. Deste modo ambas as funções tem pleno acesso as variáveis, podendo ser acessadas e modificadas por quaisquer uma das funções. Assim não é necessário a passagem de parâmetros para a função.

6.6 Recursividade

A recursividade talvez seja a mais importante vantagem das funções em C. **Recursão** é o processo pelo qual uma função chama a si mesma repetidamente um numero finito de vezes. Este recurso é muito útil em alguns tipos de algoritmos chamados de **algoritmos recursivos**.

Vejamos um exemplo *clássico* para esclarecermos o conceito: calculo do **fatorial** de um número. A definição de fatorial é:

```
n! = n . (n-1) . (n-2) . ... . 3 . 2 . 1
0! = 1
```

onde `n` é um numero inteiro positivo. Uma propriedade (facilmente verificável) dos fatoriais é que:

```
n! = n . (n-1)!
```

Esta propriedade é chamada de propriedade recursiva: o fatorial de um numero pode ser calculado através do fatorial de seu antecessor. Ora, podemos utilizar esta propriedade para escrevermos uma rotina recursiva para o calculo de fatoriais. Para criarmos uma rotina recursiva, em C, basta criar uma chamada a própria função dentro dela mesma, como no exemplo a seguir.

Programa exemplo: No arquivo `e0606.cpp` existe uma rotina recursiva para o calculo de fatoriais.

Uma função recursiva cria a cada chamada um novo conjunto de variáveis locais. Não existe ganho de velocidade ou espaço de memória significativo com o uso de funções recursivas. Teoricamente uma algoritmo recursivo pode ser escrito de forma iterativa e vice-versa. A principal vantagem destes algoritmos é que algumas classes de algoritmos [de inteligência artificial, simulação numérica, busca e ordenação em arvore binaria, etc.] são mais facilmente implementadas com o uso de rotinas recursivas. O estudo deste tipo de algoritmo está, porém, além do alcance deste texto.