Capítulo 1

Funções

" A arte de programar consiste em organizar e dominar a complexidade. "

- Edsger W. Dijkstra -

Sumário:

1.1- Funções

Entendemos por função um segmento de código com uma estrutura muito semelhante à de um programa, que tem por finalidade resolver um problema específico.

As funções são utilizadas para separar um programa em partes logicamente coerentes e, devem possuir as seguintes propriedades: Realizar uma única tarefa; o seu código deve ser o mais independente e genérico possível; possuir um mecanismo de comunicação com o exterior.

Quando um programa chama (activa ou invoca) uma função para realizar uma tarefa específica, este não tem de preocupar-se como essa tarefa será realizada. O programa apenas envia os dados para a função e, espera que esta envie

uma solução correcta. Isto quer quizer que o funcionamento correcto de um programa depende apenas do funcionamento correcto das funções que o compoêm.

Toda a função possui a seguinte sintaxe:

```
tipo nome (declaração de parâmetros)
  {
     Variáveis locais;
     Comandos da função;
     Valor de retorno;
  }
```

Toda a função tem um nome que deve ser escrito respeitando os princípios de declaração de variáveis. O nome identifica de forma unívoca cada função num programa e, se pode ter duas funções com o mesmo nome.

Qualquer função pode receber dados do exterior e esses dados são recebidos pelos parâmetros. Mas, uma função só pode enviar para o exterior um e apenas um valor. Esse envio é feito pelo valor de retorno.

Antes do nome de uma função devemos identificar o tipo de dados que essa função retorna.

Exemplo 1: desenvolva um programa para converter a temperatura de graus Celsius para graus Fahrenheit.

Podemos dividir este problema em duas funções. A primeira fica responsável pela conversão da temperatura, enquanto a segunda responsável pela leitura da temperatura em graus Celsius e a impressão da correspondente temperatura em graus Fahrenheit. Desta forma cada funcionalidade do problema é implementada por uma função independente. O código que implementa a função que faz a conversão é descrito a seguir:

```
float celsiusParaFahrenhei (int tc)
{
  float tf;
  f = 1.8 * tc + 32;
  return tf;
}
```

A grande vantagem de dividir o programa em pequenas funções é aumentar a sua consistência e a sua reutilização de código.

Depois de termos a certeza que esta função retorna para qualquer temperatura em gruas Celcius a correspondente temperatura em graus Farhrenheit correcta, podemos utiliza-la em qualquer programa que necessita dessa conversão, minimizando desse modo o custo de implementação do programa.

Com este tipo de programação, o código da função principal fica mais simples, ele deve ser genérico, porque os detalhes da solução, foram executados pelas funções. O código que implementa a função principal é descrito a seguir:

```
int main ( )
{
   float cels, fahr ;
   printf ("Entre com temperatura em Celsius : " ) ;
   scanf ("%f " , &cels ) ;
   fahr = celsiusParaFahrenhei (cels );
   printf ("Temperatura em Fahrenheit : %f " , fahr );
}
```

1.2 - Parâmetros e Valor de Retorno

Uma função deve ter uma interface bem definida. Quando tivermos de implementar uma função, devemos definir de uma forma clara, o que a função importa via parâmetros e o que a função exporta pelo valor de retorno.

Para além disso, também devemos definir no cabeçalho da função o tipo de cada parâmetro e o tipo de dados do valor que a função exporta. O tipo de dados que a função exporta é que determina o tipo de dados da função e não o contrário.

Uma função pode importar zero ou mais valores pelos parâmetros, mas exporta um e apenas um valor pelo retorno da função.

Exemplo 1: desenvolva uma função para calcular o volume de um cilindro.

Pela geometria plana, sabemos que o volume de um cilindro é determinado pela formula $v = \pi r^2 h$. Uma possível implementação é descrita pela seguinte função:

```
#define PI 3.14159

float volumeCilindro ( float r , float h)
{
  float v ;
  v = PI *r*r *h ;
   return v ;
}
```

Para testar esta função, podemos escrever uma função principal. Nessa função principal, os valores do raio e da altura são capturados pelo teclado e converter a temperatura e consiste em ler os valores do raio e altura, invoca-se a função para calcular o volume e em seguida, mostra-se essa informação na tela. O código que implementa essa função é descrito a seguir:

```
int main ()
{
  float raio , altura , volume;
  printf ( "Entre com o valor do raio : " );
  scanf ( "%f " , &raio ) ;
  printf ( "Entre com o valor da al tura : " );
  scanf ( "%f " , &altura );
```

```
    volume = volumeCilindro ( raio , altura ) ;
    printf ( "Volume do cilindro = %f " , volume ) ;
    return 0;
}
```

Observe que os valores passados na chamada da função devem corresponder aos parâmetros em número e tipo. Neste caso, a função volumeCilindro espera receber dois parâmetros do tipo real (float).

Na chamada da função são enviados dois valores para a função volumeCilindro. Esses valores são os conteúdos das variáveis raio e altura, que também são do tipo real (float). O retorno da função é armazenado no variável volume, que é o calculo da fórmula $v = \pi r^2 h$.

Observe ainda que a chamada de uma função que tem um valor de retorno associado é uma expressão que, quando avaliada, resulta no valor retornado. Assim, uma chamada de função pode aparecer dentro de uma expressão maior. Por exemplo, se quiséssemos calcular o volume de metade do cilindro, poderíamos ter escrito:

```
volume = volumeCilinbro ( raio , al tura ) / 2.0;
```

Para terminar é importante salientar que na chamada de uma função, passa-se valores para os parâmetros. Isso quer dizer que podemos utilizar uma expressão como parâmetro, desde que o resultado dessa expressão coincida com o tipo do parâmetro. Por exemplo, poderíamos calcular a área de um cilindro com o dobro da altura, fazendo:

```
volume = volumeCilindro ( raio ,2* altura ) ;
```

1.3 - Procedimentos

Entendemos por procedimentos, funções que não exportam nenhum valor via valor de retorno. Na linguagem C os procedimentos são conhecidos como **funções do tipo void** e possuem a seguinte sintaxe:

```
void nome (declaração dos parâmetros)
{
    Declaração dos identificadores locais;
    Comandos da função;
}
```

Nas funções do tipo void terminam quando encontrarmos o delimitador de fim de blocos.

Exemplo 1: desenvolva uma função para imprimir uma determinada temperatura.

Esta função que denominamos por imprimirTemp (), recebe como parâmetro uma temperatura em graus Celcius, Farhrenheit ou Kelvin, e mostra na tela essa informação.

```
void printTemp( int t)
{
   printf ( "%f", t);
}
```

Agora podemos reutilizar essa função para implementar uma função principal que lê a temperatura em graus Celcius e mostra da tela a correspondente temperatura em graus Farhrenheit.

```
int main ()
{
   float cel, fahr;
   printf ("\n Entre com temperatura em Celsius : " );
   scanf ("%f " , &cel);
   fah = celsiusParaFahrenheit (cels );
   printf ("\n Temperatura em Fahrenheit : " );
   printTemp (fahr);
   return 0;
 }
```

1.4 - Passagem de Parâmetros

A forma mais eficiente de comunicação de uma função com o exterior é através da passagem de parâmetros. Em qualquer linguagem de programação, temos duas formas de passagem de parâmetros, a passagem por valor mais frequente e a passagem por referência.

Na **passagem por valor** uma cópia dos parâmetros da chamada é transferida para a função. Logo, os parâmetros da chamada e os parâmetros da função estão armazenados em posições de memória diferentes. Então qualquer alteração aos parâmetros da função, não altera o valor da variável que será transferida no exterior da função.

Exemplo 1: desenvolva uma função para a potência de expoente inteiro, x^n para n > 0.

Pela matemática elementar, sabemos que $x^0 = 1$ e que $x^n = x*x*x*….*x$, ou seja, multiplicamos x, n vezes.

Esta função, denominada por potencia (), recebe como parâmetro o valor da base e do expoente. Retorna o valor de potencia da base elevada ao expoente. Uma possível implementação é descrita a seguir:

```
float potencia (float base, int exp)
{
  float pot ;
   int i;
   pot = 1.0;
```

```
    for (i = 0 ; i < exp ; i++)
        pot = pot * base ;
    return pot;
}
```

Para testar esta função, podemos escrever uma função principal que segue o mesmo padrão das funções principais anteriores. Os valores da base e do expoente são capturados pelo teclado, se o expoente for positivo, calcular o valor da potência e mostrar na tela essa informação, no caso contrário, mostrar uma mensagem de erro adequada. Um possível código que implementa essa função é descrito a seguir:

```
int main ()
{
   int y;
   float x;
   printf ("\n Entre com o valor do expoente : " );
   scanf ("%d " , &y);
   if ( y >= 0)
     {
       printf ("\n Entre com o valor da base : " );
       scanf ("%d " , &x);
       pot = potencia(x , y);
       printf ("\n potencia de %f elevado a %d = %f “, pot );
       return 0;
     }
   else
       printf (“\n expoente invalido “);
    return 0;
}
```

Sempre que for possível utilize a passagem por valor, mas quando tivermos a necessidade de devolver mais do que um valor, devemos utilizar a passagem por referência.

Na **passagem por referência**, a comunicação entre os parâmetros da função e os parâmetros da chamada são feitas por endereços. Isso quer dizer que os parâmetros da função e os parâmetros da chamada compartilham as mesmas posições de memória. Então, qualquer alteração aos parâmetros da função afectam as correspondentes variáveis no exterior da função. Para efectuar essa passagem devemos seguir as seguintes regras: na chamada da função os parâmetros passados por referência devem possuir como prefixo o operador de endereço (&), enquanto na declaração da função os correspondentes parâmetros devem possuem como prefixo o operador de acesso indirecto (*).

Exemplo 2: desenvolva uma função para trocar o conteúdo de duas variáveis.

Esta função, denominada por trocar (), recebe como parâmetro os valores de x e de y. Devolve os mesmos parâmetros com o seu conteúdo trocado. A

implementação desta função baseia-se em declarar os parâmetros x e y como ponteiros, ou seja:

```
void trocar ( int *x , int *y)
{
   int aux;
   aux = *x;
  *x = y*
  *y = aux;
 }
```

Agora podemos reutilizar essa função para implementar uma função principal para ler dois valores inteiros, armazena-los nas variáveis x e y, trocar o seu conteúdo e mostrar em seguida o conteúdo dessas variáveis na tela. Uma possível implementação consiste em declarar os parâmetros da chamada como endereços.

```
int main()
{
  int x, y;
  printf (" \n  Entre com o valor de x : ");
  scanf (" %d ", &x);
  printf (“ \n Entre com o valor de y : “);
  scanf ( “ %f ”, &y);
  trocar (&x,&y);
  printf ( “\n  Valor de x = %d “, x);
  printf ( “\n  Valor de y = %d “, y);
  return 0;
}
```

1.5 - Funções Generalizadas

Vimos que as funções importavam vários valores pelos parâmetros, e exportavam um e apenas um valor pelo valor de retorno.

Mas existem alguns problemas de científicos e comerciais, que temos a necessidade de devolver para o exterior vários valores. Para colmatar essa restrição a linguagem C possui a função generalizada.

Uma função generalizada, é um subprograma que recebe vários valores pelos parâmetros, devolve pelos mesmos parâmetros vários valores e retorna um e apenas um valor pelo valor de retorno.

Mas, para realizar essa comunicação á função generalizada utiliza a passagem por referencia enquanto a função clássica (simples) utiliza apenas a passagem por valor.

Na passagem por referencia os parâmetros da função são precedidos pelo operador de endereço (*), enquanto que na chamada dessa função, os parâmetros são precedidos pelo operador de endereço. (&)

Exemplo 1: desenvolva uma função para calcular as raízes reais de uma equação do segundo grau $ax^2+bx+c = 0$.

Esta função, recebe como parâmetros, os coeficientes da equação e devolve como parâmetros as raízes dessa equação. Mas, como existem vários casos particulares, vamos retornar como valor de retorno, uma informação a notificar o tipo de raiz encontrada. Esse tipo de raiz é determinado pela tabela:

1 = Quando há duas raízes diferentes;
0 = Quando há duas raízes iguais;
-1 = Quando não há raízes reais;
-2 = Quando a equação do primeiro grau;
-3 = Quando não é uma equação.

```
int equacao2Grau (double a, double b, double c, double *x1, double *x2)
{
   double delta;
   if (a == 0.0 && b == 0.0)            /* não é uma equação*/
      return -3;
   if (a == 0.0)                        /*equação do 1. Grau */
      {
       *x1 = -c / b;
        return -2;
      }
   delta = b*b - 4*a*c;
   if (delta < 0)                        /* se delta < 0, raízes não são reais */
      return -1;
   if (delta == 0)                      /* se delta = 0, raízes reias iguais */
      {
        *x1 = *x2 = -b / (2*a);
        return 0;
      }
   *x1 = (-b + sqrt(delta))/(2*a);       /* se delta > 0, raízes reais diferentes */
   *x2 = (-b - sqrt(delta))/(2*a);
   return 1;
}
```

É na função principal, que teremos o cuidado de tratar o tipo da raiz e notificar e mostrar na tela mensagem apropriada. Isso quer dizer, que o interface do programa deve ser feito em duas camadas. Na primeira camada temos as entradas e saídas de dados feitos pelas funções **scanf**() e **printf**(). Esse interface deve ser reservada a função principal.

Nas funções que nós desenvolvemos, a entrada de dados é feita pelos parâmetros enquanto a saída deve ser feita pelo valor de retorno.

Isso quer dizer que normalmente, note que está escrito normalmente, não se deve colocar a função **scanf**() e a função **printf**() nas funções que nós desenvolvemos.

Apresentamos em seguida, o código que implementa a função principal:

```
#include <stdio.h>
#include <stdlib.h>
int main ( )
{
   float a, b, c;
   int  codRaiz;
   printf (“\n Entre com o coeficiente a:“);
   scanf  (“% s”, &a);
   printf (\n  entre com o coeficiente b );
   scanf (“% s”, &b);
   print (“\n Entre com o coeficiente c “);
   scanf (“% s”, &c);
   codRaiz = equacao2Grau ( a,  b,  c, &raiz1, , &raiz2);
   switch ( codRaiz)
   {
       case '-3 ':
           printf ("\n nao equacao do segundo grau”);
           break;
       case '-2:
          printf ("\n  equacao do primeiro grau”);
          break;
       case '-1 ':
           printf ("\n  nao ha raizes  reais ”);
           break;
       case '0 ':
           printf ("\n  raizes  reais iguais ”);
           printf ( “ % f     % f “ raiz1, raiz2);
           break;
       case '1 ':
           printf ("\n  raizes  reais diferentes ”);
           printf ( “ % f     % f “ raiz1, raiz2);
           break;
    }
   system (“pause”);
   return 0;
}
```

1.6 - Recomendações Bibliográficas

Para o leitor aprofundar os seus conhecimentos sobre a linguagem C, recomendamos os seguintes livros:

Damas L.;- *Linguagem C*, 9º Edição, FCA, 1999, Lisboa, Portugal

Deitel H. M., Deitel, P. J.;- *C How to Program,* 8 th Edition, Pearson Education,

Kelly A., Pohl I. ;- *A Book on C*, 4nd Edition, Addison Wesley, 1998.

King K. N.;- *C Programming: A Modern Approach*, 2nd Edition, Norton & Company, 2008.

Mizrahi V. V.;- *Treinamento em Linguagem C*, 2ª Edição, Pearson Education, 2008, Brasil

1.7 - Exercícios Propostos

1.7.1-Desenvolva uma função para receber dois inteiros positivos a e b e verificar se b corresponde aos últimos dígitos de a. Por exemplo:

a	**b**	**Está Contido**
567890	890	Sim
1243	1243	Sim
2457	245	Não
4571	12457	Não

Utilize esta função para desenvolver um programa que lê dois números inteiros positivos a e b e verifica se o menor deles é um subnúmero do maior. Por exemplo:

a	**b**	
567890	678	b é subnúmero de a
1243	2212435	a é subnúmero de b
235	236	Um não é subnúmero do outro

1.7.2-Desenvolva uma função que recebe dois números inteiros com quatro algarismos e retorna 1 se esses números são inversos e zero no caso contrário. Dois números são inversos se os dígitos do primeiro estão na ordem inversa no segundo.

1.7.3-Desenvolva uma função que recebe um número real e retorne o valor desse número arredondado com base na seguinte tabela. Se casa decimal estiver entre 0.0 a 0.24 arredondar para 0.25; entre 0.25 a 0.54 arredondar para 0. 55; entre 0.56 a 0.74 arredondar para 0.75; entre 0.75 a 0.99 arredondar 1.0.

1.7.4-Desenvolva uma função que recebe um número real x e um número inteiro n. Retorna à potência de x elevado a y, mas para realizar esse calculo utilize apenas multiplicações sucessivas.

1.7.5-Dois números inteiros positivos com dois algarismos são pares combinados se forem pares e se os dígitos de um ocorrem na ordem inversa do outro. Por exemplo, os números 48 e 84 são pares combinados, enquanto os números 23 e 32 não são. Desenvolva uma função que recebe dois números inteiros positivos com dois dígitos e verifica se eles são pares combinados.

1.7.6-Cada termo da sequência de Tribonacci é dado pela soma dos três termos anteriores. Supondo que a sequência inicia com os números T_1= 1, T_2 =1 e T_3 = 2, os 8 restantes termos são os números inteiros:

1, 1, 2, 4, 7, 13, 24, 44, 81, 149, ….

Desenvolva uma função que recebe um número inteiro positivo n > 0 e os três termos iniciais de Tribonacci. Calcule e retorne o valor de Tribonacci.

1.7.7-Desenvolva uma função que recebe as coordenadas de um ponto no plano cartesiano (X,Y). Retorne em que quadrante esse ponto se encontra. Se o ponto estiver no interior do quadrante retorne um número entre 1 a 4, se estiver sobre um dos eixos, retorne o valor -1, se estiver na origem retorne o valor zero.

1.7.8-Dizemos que um número inteiro n pode ser partido em dois números inteiros n1 e n2 se n pode ser escrito da forma n1n2, isto é, concatenar n1 com n2. Desenvolva uma função que recebe três inteiros positivos n, n1 e n2. Verifique se n pode ser partido em dois números inteiros n1 e n2. Vamos supor que esses números não contêm o dígito 0. Por exemplo: para n = 123456, n1 = 12 e n2 =3456, a resposta é sim, mas para n = 123456, n1 = 456 e n2 =123, a resposta é não.

1.7.9-Dados dois números inteiros positivos x e y. Desenvolva uma função para calcular o mdc (x, y) para x > y pelo algoritmo de Euclides.