O que é função

Uma **função** é um pedaço de código que faz alguma tarefa específica e pode ser chamado de qualquer parte do programa quantas vezes desejarmos.

Podemos também dizer que funções agrupam operações em um só nome que pode ser chamado em qualquer parte do programa. Essas operações são então executadas todas as vezes que chamamos o nome da função.

Utilizamos funções para obter:

- **Clareza do código**: separando pedaços de código da função main(), podemos entender mais facilmente o que cada parte do código faz. Além disso, para procurarmos por uma certa ação feita pelo programa, basta buscar a função correspondente. Isso torna muito mais fácil o ato de procurar por erros.
- **Reutilização**: muitas vezes queremos executar uma certa tarefa várias vezes ao longo do programa. Repetir todo o código para essa operação é muito trabalhoso, e torna mais difícil a manutenção do código: se acharmos um erro nesse código, teremos que corrigi-lo em todas as repetições do código. Chamar uma função diversas vezes contorna esses dois problemas.
- **Independência**: uma função é relativamente independente do código que a chamou. Uma função pode modificar variáveis globais ou ponteiros, mas limitando-se aos dados fornecidos pela chamada de função.

A ideia funções é permitir você encapsular várias operações em um só escopo que pode ser invocado ou chamado através de um nome. Assim é possível então chamar a função de várias partes do seu programa simplesmente usando o seu nome.

Exemplo:

```
#include <stdio.h>
int main(){
 imprime_par(3,4);
 imprime_par(-2,8);
 return 0;
}
```

No exemplo acima, a função **imprime_par** foi usada para executar o pedaço de programa que imprime um par de números. A saída do programa acima será:

```
{3,4}
{-2,8}
```

A função **imprime_par** é definida da seguinte forma:

```
void imprime_par(int a, int b){
   printf("{ %d, %d }\n",a,b);
}
```

O programa completo em C é mostrado abaixo:

```c
#include <stdio.h>

/**
 * Declaração da função imprime_par
 * Essa função recebe dois inteiros como argumento e os imprime
 * da seguinte forma {a,b}
 */
void imprime_par(int a, int b);


int main(){
    imprime_par(3,4);  //chamando a função
    imprime_par(-2,8); //chamando novamente
    return 0;
}

//Implementação da função
//A implementação da função pode conter várias linhas de código
void imprime_par(int a, int b){
   printf("{ %d, %d }\n",a,b);
}
```

A definição de funções em C deve ser feita antes do uso das mesmas. Por isso em nosso exemplo definimos a função **imprime_par** antes de usá-la dentro do main.

A linha que define ou declara a função também é conhecida como **assinatura** da função. Normalmente as assinaturas das funções são definidas dentro de arquivos de cabeçalho **.h**

Definindo uma função

Uma função pode necessitar de alguns dados para que possa realizar alguma ação baseada neles. Esses dados são chamados **parâmetros** da função. Além disso, a função pode retornar um certo valor, que é chamado **valor de retorno**. Os parâmetros (e seus tipos) devem ser especificados explicitamente, assim como o tipo do valor de retorno.

A forma geral da definição de uma função é:

```
 [tipo de retorno da função] [nome da função] (1° parâmetro, 2° parâmetro, …)
 {
    //código
 }
```

- Para o nome da função e dos parâmetros valem as mesmas regras que foram dadas para os nomes de variáveis. Não podemos usar o mesmo nome para funções diferentes em um programa.
- Todas as funções devem ser definidas antes da função **main**, ou deve ser feito o **protótipo** da função, que veremos mais adiante.
- O código deve estar obrigatoriamente dentro das chaves e funciona como qualquer outro bloco.

Valor de retorno

Freqüentemente, uma função faz algum tipo de processamento ou cálculo e precisa retornar o resultado desse procedimento. Em C, isso se chama **valor de retorno** e pode ser feito com a instrução **return**. Para poder retornar um valor, precisamos especificar seu tipo (char, int, float, double e variações). Para efetivamente retornar um valor, usamos a instrução return seguida do valor de retorno, que pode ou não vir entre parênteses. Um exemplo bem simples de função que retorna um valor inteiro:

```
int tres(){
   return 3;   // poderia também ser return (3);
}
```

O tipo de retorno, além dos tipos normais de variáveis (char, int, float, double e suas variações), pode ser o tipo especial **void**, que na verdade significa que não há valor de retorno.

Nota Muitos livros dizem que a função main tem tipo de retorno **void**, o que não está correto. Segundo o padrão da linguagem C, a função main deve ter retorno do tipo **int**. Compiladores como o gcc darão mensagens de erro caso a função main() não seja definida corretamente.

Parâmetros

Como já foi dito, um parâmetro é um valor que é fornecido à função quando ela é chamada. É comum também chamar os parâmetros de **argumentos**, embora argumento esteja associado ao valor de um parâmetro.

Os parâmetros de uma função podem ser acessados da mesma maneira que variáveis locais. Eles na verdade funcionam exatamente como variáveis locais, e modificar um argumento não modifica o valor original no contexto da chamada de função, pois, ao dar um argumento numa chamada de função, ele é copiado como uma variável local da função. A única maneira de modificar o valor de um parâmetro é usar ponteiros, que serão introduzidos mais adiante.

Para declarar a presença de parâmetros, usamos uma *lista de parâmetros* entre parênteses, com os parâmetros separados por vírgulas. Cada declaração de parâmetro é feita de maneira semelhante à declaração de variáveis: a forma geral é tipo nome. Por exemplo:

```
int funcao (int a, int b)
float funcao (float preco, int quantidade)
double funcao (double angulo)
```

Para especificar que a função não usa nenhum parâmetro, a lista de parâmetros deve conter apenas a palavra-chave **void**. No entanto, ela é freqüentemente omitida nesses casos. Portanto, você poderia escrever qualquer uma destas duas linhas:

```
void funcao (void)
void funcao ()
```

Note que os nomes dos parâmetros são usados apenas na própria função (para distinguir os argumentos); eles não têm nenhuma relação com as variáveis usadas para chamar a função.

Chamadas de funções

Para executar uma função, fazemos uma **chamada de função**, que é uma instrução composta pelo nome da função, seguido pela lista de argumentos entre parênteses:

```
nome_da_função (arg1, arg2, arg3, ...);
```

Os argumentos podem ser qualquer tipo de expressão: podem ser variáveis, valores constantes, expressões matemáticas ou até mesmo outras chamadas de função.

Lembre que você deve sempre dar o mesmo número de argumentos que a função pede. Além disso, embora algumas conversões de tipo sejam feitas automaticamente pelo compilador, você deve atender aos tipos de argumentos.

Note que o valor dos argumentos é **copiado** para a função, de maneira que as variáveis originais ficam inalteradas mesmo que na função tentemos alterá-las. A isso chamamos passagem de argumentos *por valor* (ao contrário de *por referência*). Veremos como modificar as variáveis originais na seção Ponteiros.

A própria chamada de função também é uma expressão cujo valor é o valor de retorno da função, bastando colocá-la no lado direito de um sinal de igual para guardar o valor numa variável. Por exemplo, se a função "quadrado" retorna o quadrado de um número inteiro, podemos fazer assim para calcular o quadrado de 11 na variável *x*:

```
int x = quadrado (11);
```

Dois exemplos[editar | editar código-fonte]

```
#include <stdio.h>

int quadrado (int x){
   return (x * x);
}

void saudacao (void){
   printf ("Olá!\n");
}

void despedida (void){
   printf ("Fim do programa.\n");
}

int main (){
   int numero, resultado;
   saudacao ();

   printf ("Digite um número inteiro: ");
   scanf ("%d", &numero);
   resultado = quadrado (numero);
   printf ("O quadrado de %d é %d.\n", numero, resultado);

   despedida ();
   return 0;
}
```

Você veria na tela, ao executar o programa:

```
Olá!
Digite um número inteiro: 42
O quadrado de 42 é 1764.
Fim do programa.
```

Repare que, ao chegar na chamada de uma função, o programa passa o controle para essa função e, após seu término, devolve o controle para a instrução seguinte na função original.

Mais um exemplo, com uma função de 3 argumentos:

```c
#include <stdio.h>

/* Multiplica 3 numeros */
void mult (float a, float b, float c){
   printf ("%f",a*b*c);
}
 int main (){
   float x, y;
   x = 23.5;
   y = 12.9;
   mult (x, y, 3.87);
   return 0;
}
```

Protótipo ou Declaração de função

Quando um programa C está sendo compilado e uma chamada de função é encontrada, o compilador precisa saber o tipo de retorno e os parâmetros da função, para que ele possa manipulá-los corretamente. O compilador só tem como saber isso se a função já tiver sido definida. Portanto, se tentarmos chamar uma função que está definida abaixo da linha onde estamos fazendo a chamada, ou mesmo em outro arquivo, o compilador dará uma mensagem de erro, pois não conseguiu reconhecer a função.

```
//Exemplo de erro de chamada de função
int main(){
  int a = 1;
  int b = 2;
  soma(a,b); // erro: a função está definida abaixo desta linha!
}
 void soma(int a, int b){
 printf("%d", a+b);
}
```

Nesses casos, podemos **declarar** uma função antes de defini-la. Isso facilita o trabalho de usar diversas funções: você não precisará se importar com a ordem em que elas aparecem nos arquivos.

A declaração de função (também chamada de protótipo de função) nada mais é que a definição da função sem o bloco de código. Como uma instrução, ela deve ser seguida de um ponto-e-vírgula. Portanto, para declarar a função:

```
int quadrado (int x){
    return (x * x);
 }
```

escreveríamos:

```
 int quadrado (int x);
```

Numa declaração, também podemos omitir os nomes dos parâmetros, já que estes são ignorados por quem chama a função:

```
int quadrado (int);
```

Poderíamos, por exemplo, reorganizar o início do programa-exemplo dado um pouco acima, o que permitiria colocar as funções em qualquer ordem mesmo que houvesse interdependência entre elas:

```
#include <stdio.h>

int quadrado (int x);
void saudacao (void);
void despedida (void);

// seguem as funções do programa
```

Note que a definição da função não deve contradizer a declaração da mesma função. Se isso ocorrer, uma mensagem de erro será dada pelo compilador.

Variáveis locais versus globais

Quando declaramos as variáveis, nós podemos fazê-lo

- Dentro de uma função
- Fora de todas as funções inclusive a main().

As primeiras são as designadas como locais: só têm validade dentro do bloco no qual são declaradas. As últimas são as globais, elas estão vigentes em qualquer uma das funções.

Quando uma função tem uma variável local com o mesmo nome de uma variável global a função dará preferência à variável local. Daqui conclui-se e bem que, podemos ter variáveis com o mesmo nome, o que contradiz o que nós dissemos no capitulo das variáveis.

Então reformulamos:

> Apenas na situação em que temos 2 variáveis locais é que é colocada a restrição de termos nomes diferentes caso contrário não conseguiríamos distinguir uma da outra.

"largo" e "alto" são variáveis internas fazem parte de "minhaFuncion()".

```
void minhaFuncion(){
  double largo = 5;
  double alto = 6;
}
```

As variáveis largo e alto não estão definidas aqui abaixo, isto quer dizer que elas nao tem nem um valor.

E não podemos usar os valores definido dentro da "minhaFuncion", pois não há nenhuma instrução que defina que valor usar. Lembre-se: O computador não vai adivinhar qual valor usar. Deve-se definir cada instrução.

```
void calcular(){ /*Não houve definição de valor entre parenteses*/
   long superficie = largo * alto;/*Error valor nao definido*/
   return(superficie);
}
```

Nesse exemplo abaixo, poderemos usar o valor das variáveis externas dentro de todas as funções. Exemplo:

```
 #include <stdio.h>
/* Variaveis externas */
long largo = 10;
long alto  = 20;

void F_soma (){
  /*soma é uma variavel interna
    e largo e alto sao variaveis externas */
  long soma = largo + alto ;
  printf("largo + alto = %i \n", soma);
}

long calcular(){
  long superficie = largo * alto;
  return superficie;
}
```

```
int main(){
    F_soma ();
    Printf("Superficie : %ld \n", calcular() );
    return 0;

}
```

Curiosidade A palavra reservada "auto" serve para dizer que uma variável é local, mas a utilização de **auto** não é mais necessária pois as variáveis declaradas dentro de um bloco já são consideradas locais.

Passagem de parâmetros por valor e por referência

O que nós temos feito quando chamamos uma função é a dita **chamada por valor**. Quer dizer, quando chamamos uma função e passamos parâmetros para a *função protótipo* e depois para a *função definição*, o valor dos argumentos passados são copiados para os parâmetros da função. Estes existem independentemente das variáveis que foram passadas. Eles tomam apenas uma cópia do valor passado, e se esse valor for alterado o valor dos argumentos passados não são alterados. Ou seja, não são alterados os valores dos parâmetros fora da função. Este tipo de chamada de função é denominado chamada (ou passagem de parâmetros) **por valor**.

Dito de outra maneira. Passamos a variável “a”, ela entra na definição da função como cópia de “a” e entra como variável “b”. Se a variável “b” for alterada no decorrer da função, o valor de “a” não é alterado.

```c
#include <stdio.h>
float quadrado(float num);      //protótipo da função quadrado()
int main (){
    float num, res;//declara 2 variáveis: num , res
    printf("Entre com um numero: ");
    scanf("%f", &num);    //associa o valor lido a variável num
    res = quadrado(num);//chama a função quadrado e passa o parâmetro num
    printf("\n\nO numero original e: %f\n", num);
    printf("e seu quadrado vale: %f\n", res);
    getchar();
    return 0;
}
float quadrado (float num){     //descrição da função quadrado
      return num * num; //retorna num ao quadrado
}
```

Quando a função main() é executada, ela chega a meio e vê uma chamada para a função quadrado() e onde é passado o parâmetro "num". Ela já estava a espera, pois "viu" o protótipo. Ela então vai executar a função que está depois da função do **main**(). E o que acontece é que o "num", vai ficar com o dobro do valor. Esse valor do **main**() vai entrar novamente no **main**(). E é associado á variável "res". Depois temos a impressão da variável "num" e "res". Ora o que acontece é que o valor do "num" fica igual ao valor antes de entrar na função. Fazemos a mesma coisa agora com a variável "a" e "b", e vemos que agora a função a é alterada. Resumindo, o valor variável quando entra numa outra função não é alterado (na passagem por valor).

Quando o valor do parâmetro é alterado denominamos chamada (ou passagem) **por referência**. O C não faz chamadas por referência. Mas podemos simular isto com outra arma do C que são os **ponteiros**, que serão melhor explicados mais adiante.

void

Como dissemos, uma função retorna um valor. E pode receber parâmetros. O void é utilizado da seguinte forma:

```c
void função(void){
  //codigo
}
```

No exemplo acima, a palavra **void** define que:

- não vai receber parâmetros; e
- não vai retornar qualquer valor.

Ou melhor, **void** é uma explicitação do programador que aquela função não vai receber ou retornar nenhum valor.

O valor da função é ignorado, mas a função realmente retorna um valor, por isso para que o resultado não seja interpretado como um erro e bom declarar void.

Nota

> Não se pode utilizar **void** na função principal **main**, apesar de existirem exemplos com void em algumas bibliografias. Infelizmente, alguns compiladores aceitam void main(). O main() é especial e tem de retornar um int. Uma execução bem sucedida do programa costuma retornar 0 (zero) e, em caso de erro, retorna 1 (um).

```c
/*
1 - Criar um programa em C que receba dois números inteiros e um caracter
indicando uma operação aritmética (+ / - *), e que mostre o resultado da
operação, ou uma mensagem de erro se a operação não puder ser realizada.
O programa deve possuir quatro funções, uma para cada operação aritmética
a ser realizada, sendo:
  + sem parâmetro e sem retorno
  / sem parâmetro e com retorno
  - com parâmetro e sem retorno
  * com parâmetro e com retorno
*/
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>

void soma();              // + sem parâmetro e sem retorno
int divisao();            // / sem parâmetro e com retorno
void sub(int,int);        // - com parâmetro e sem retorno
void sub1(int,int,int*); // - com parâmetro e sem retorno. O retorno é simulado
                          //     com o terceiro parâmetro, que é passado à função por
                          //     referência
int mult(int,int);        // * com parâmetro e com retorno

int sa, sb; //são variáveis globais. Neste programa utilizadas nas funções
            //soma() e divisao(), que não possuem parâmetros

/*
  a linha 44 declara uma variável que é um ponteiro para inteiro, que está
    sendo inicializada com NULL (int *a = NULL). Se esta variável for declarada
    desta forma, a passagem de parâmetro para a função sub1, na linha 69 deve
    ser escrita da seguinte forma (sub1(x,y,a);), uma vez que o conteúdo da
    variável a já é um endereço de memória, e o printf da linha 70 deve ser
    escrito assim (printf("\n..A subtração é %d",*a);), para que o conteúdo
    para o qual a variável a aponta seja impresso.
    Se a variável não for declarada como um ponteiro (int a;), a passagem de
    parâmetro da linha 69 deve ser realizada com o envio do endereço de memória
    da mesma (sub1(x,y,&a);), e o printf da linha 70 deve mostrar diretamente
    o valor da variável (printf("\n..A subtração é %d",a);)
*/
int main(){
  setlocale(LC_ALL,"Portuguese");
  char op;
  int x, y;
  int *a = NULL; // OU int a; //continua na linha 53
  printf("\nCalculadora para 4 operações básicas: + / - *\n");
  printf("Digite o primeiro valor inteiro: ");
  scanf("%d",&x);
  printf("Digite o segundo valor inteiro: ");
  scanf("%d",&y);
  printf("Informe a operação que será realizada: ");
  fflush(stdin);
  op = getchar();
  switch (op){
     case '+':
       sa = x; //sa --> global
       sb = y; //sb --> global
       soma();
       break;
     case '/':
       sa = x; //sa --> global
       sb = y; //sb --> global
       if (divisao() == 'a')
         printf("\nErro de divisão por zero\n");
         else
           printf("A divisao e %d", divisao());
       break;
     case '-':
```

PONTEIRO

```
        sub(x,y);
        sub1(x,y,a); //OU sub1(x,y,&a)  //ver linha 28
        printf("\n..A subtração é %d",a); //OU printf("\n..A subtração é %d",a);
        break;
      case '*':
        printf("\nA multiplicao e %d",mult(x,y));
        break;
        default:
        printf("\nOperação inválida\n");
    }
    return 0;
  }
  //**** Função soma ************************************************************
  void soma() /*Esta função não possui parâmetros e também não possui retorno
                O resultado da função é escrito pela própria função
                São utilizadas variáveis globais ao invés de parâmetros para a função */
  {
    printf("\nA soma e %d",sa + sb);
  }
  //**** Função divisão *********************************************************
  int divisao() /*Esta função não possui parâmetros, mas possui retorno
                  O resultado da função é retornado pelo comando <return 'a';>
                  São utilizadas variáveis globais ao invés de parâmetros para a função*/
  {
    if (sb != 0)
      return sa / sb;
      else
        return 'a';
  }
  //**** Função sub *************************************************************
  void sub(int a, int b) /*Esta função possui parâmetros (a e b) que são passados
                           por valor
                         A função não possui retorno, e o seu resultado é escrito
                           pela própria função  */
  {
    printf("\nA subtracao e %d",a - b);
  }
  //**** Função sub1 ************************************************************
  void sub1(int a, int b, int *x) /*Esta função possui parâmetros (a e b) que são
                                    passados por valor (*x) que é passado por
                                    referência (*x é um ponteiro para um tipo inteiro)
                                  A função não possui retorno. O retorno é simulado
                                    pelo ponteiro *x  */
  {
    *x = a - b;
  }
  //**** Função mult ************************************************************
  int mult(int a, int b) /*Esta função possui parâmetros (a e b) que são passados por
                           valor
                         A função possui retorno, que é realizado pelo comando
                           <return (a * b);>  */
  {
    return (a * b);
  }
```