## PROGRAMAÇÃO MODULARIZADA

Sub-rotinas ou subprogramas são blocos de instruções que realizam tarefas específicas. O código de uma sub-rotina é carregado uma vez e pode ser executado quantas vezes for necessário. Dessa maneira, os programas tendem a ficar menores e mais organizados, uma vez que o problema pode ser subdividido em pequenas tarefas.

Os programas estruturados são executados linearmente, uma linha após a outra, até o fim. Entretanto, quando são utilizadas sub-rotinas, é possível a realização de desvios na execução natural do programa. Esses desvios são realizados quando uma função é chamada pelo programa principal ou por outra função.

## PRIMEIRO EXEMPLO

#include <stdio.h>

#include <stdlib.h>

void mensagem () { // Função simples: só imprime a frase Olá mundo!

printf ("Olá mundo!\n");

}

int main () {

mensagem();

printf ("Eu estou vivo!\n");

return(0);

}

Este programa mostra duas frases na tela. Foi definida uma função chamada **mensagem()** que coloca uma string na tela e retorna 0. Esta função foi chamada a partir de **main()** , que também é uma função. A diferença fundamental entre main() e as demais funções é que main é uma função especial, cujo diferencial é o fato de ser a primeira função a ser executada em um programa.

## FUNÇÕES E PROCEDIMENTOS

**Procedimentos** são rotinas (trechos ou módulos) de programas, capazes de executar uma tarefa definida pelo programador. **Funções** são rotinas similares aos procedimentos, só que retornam um valor após cada chamada. Em C não existem outras diferenças entre procedimentos e funções. A função mais próxima de um procedimento é aquela que não retorna valores para a rotina que a chamou.

Por que usar funções?

* Para permitir o reaproveitamento de código já construído (por você ou por outros programadores);
* Para evitar que um trecho de código que seja repetido várias vezes dentro de um mesmo programa;
* Para permitir a alteração de um trecho de código de uma forma mais rápida. Com o uso de uma função é preciso alterar apenas dentro da função que se deseja;
* Para que os blocos do programa não fiquem grandes demais e, por consequência, mais difíceis de entender;
* Para facilitar a leitura do programa-fonte ;
* Para separar o programa em partes (blocos) que possam ser logicamente compreendidos de forma isolada

## SEGUNDO EXEMPLO

Podemos utilizar essas duas formas de construir funções:

|  |  |
| --- | --- |
| Com a função antes de main | Com o protótipo da função antes de main |
| #include <stdlib.h>  #include <stdio.h>  **void mostraMensagem(){**  **printf(“Mensagem na função Mostra”);**  **}**  int main() {  printf(“Mensagem na função main”);  mensagem();    return(0);  } | #include <stdlib.h>  #include <stdio.h>  **void mostraMensagem() ;**  int main() {  printf(“Mensagem na função main”);  mensagem();  return(0);  }  **void mostraMensagem(){**  **printf(“Mensagem na função Mostra”);**  **}** |

## SINTAXE

 Uma função no C tem a seguinte forma geral:

*tipo\_de\_retorno nome\_da\_função (declaração\_de\_parâmetros)*   
*{*   
*corpo\_da\_função*   
*}*

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

*tipo nome1, tipo nome2, ... , tipo nomeN*

Note 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).

O corpo da função é a sua alma, e é nele que as entradas são processadas, saídas são geradas, etc.

Os **argumentos** são as entradas que a função recebe. É através dos argumentos que passamos **parâmetros** para a função. As funções printf() e scanf() são exemplos de funções que recebem argumentos.

## VARIÁVEIS GLOBAIS, LOCAIS E PARÂMETROS FORMAIS

Repetindo da Parte 1-Introdução ao C, “Há três lugares nos quais podemos declarar variáveis. O primeiro é **fora** de todas as funções do programa. Estas variáveis são chamadas variáveis globais e podem ser usadas a partir de qualquer lugar no programa. Pode-se dizer que, como elas estão fora de todas as funções, todas as funções as veem. O segundo lugar no qual se pode declarar variáveis é **no início** de um bloco de código. Estas variáveis são chamadas locais e só têm validade dentro do bloco no qual são declaradas, isto é, só a função à qual ela pertence sabe da existência desta variável, dentro do bloco no qual foram declaradas.  O terceiro lugar onde se pode declarar variáveis é na **lista de parâmetros** de uma função. Mais uma vez, apesar de estas variáveis receberem valores externos, estas variáveis são conhecidas apenas pela função onde são declaradas.”

Vamos ver as duas primeiras situações descritas nesse próximo programa:

#include <stdio.h>

#include <stdlib.h>

// variáveis globais

float media;

int a=10,b=15;

void calc\_media() {

int soma = 0; // variável local

soma = a+b;

media = soma/2.0;

}

int main(){

calc\_media();

printf("Media = %.2f", media);

return(0);

}

Já os **parâmetros formais** são declarados como sendo as **entradas de uma função**. É simples: o parâmetro formal é  uma variável local da função. Você pode também alterar o valor de um parâmetro formal, pois esta alteração não terá efeito na variável que foi passada à função. Isto tem sentido, pois quando o C passa parâmetros para uma função, são passadas apenas cópias das variáveis. Isto é, os parâmetros formais existem independentemente das variáveis que foram passadas para a função. Eles tomam apenas uma cópia dos valores passados para a função.

## #include <stdio.h>

## #include <stdlib.h>

## void calc\_triplo(int x) {

## x = x\*3;

## printf("O triplo = %d\n", x);

## }

## int main(){

## int a=10;

## calc\_triplo(a);

## return(0);

## }

## Note que no programa acima foi definido um parâmetro formal x do tipo inteiro. Esse parâmetro recebe o valor do argumento a quando a função calc\_triplo é chamada. Assim, x passa a ter o valor que havia em a e x passa a ser utilizado na função.

**ATENÇÃO:** Evite *ao máximo* o uso de variáveis globais. Elas ocupam memória o tempo todo (as locais só ocupam memória enquanto estão sendo usadas) e tornam o programa mais difícil de ser entendido e menos geral.

## PROTÓTIPOS DE FUNÇÕES

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. 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 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.

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?

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);*

onde o tipo-de-retorno, o nome-da-função e a declaração-de-parâmetros são os mesmos que você pretende usar quando realmente escrever a função. Repare que os protótipos têm uma nítida semelhança com as [declarações de variáveis](../../Sistemas%20de%20Informação/Linguagem%20C/Programacao%20C/Tutorial%20UFMG/aulas/c330.html). Vamos implementar agora um do exemplo com protótipos:

#include <stdio.h>

#include <stdlib.h>

**float quadrado(float num);**

int main () {

printf ("Informe um numero: ");

scanf ("%f",&num);

float quad = quadrado(num);

printf ("\n\nO seu quadrado vale: %.2f\n", quad);

return 0;

}

**float quadrado(float num){**

**return(num \* num);**

**}**

Observe que a função **Quadrado()** está colocada depois de **main()**, mas o seu protótipo está antes. Sem isto este programa não funcionaria corretamente.

Usando protótipos você pode construir funções que retornam quaisquer tipos de variáveis. É bom ressaltar que funções podem também retornar ponteiros sem qualquer problema. Os protótipos não só ajudam o compilador. Eles ajudam a você também: usando protótipos, o compilador evita erros, não deixando que o programador use funções com os parâmetros errados e com o tipo de retorno errado, o que é uma grande ajuda!

## USAR OU NÃO O RETORNO DE UMA FUNÇÃO?

## As funções podem retornar valores de testes, de cálculos, leituras, ... que podem ou não ser utilizados por quem chamou. Aqui temos duas formas de chamar a função calc\_media e duas formas de retornar valores:

|  |  |
| --- | --- |
| #include <stdio.h>  #include <stdlib.h>  void calc\_media(int a, int b) {  float m = (a+b)/2.0;  printf("%.2f", m);  }  int main() {  int a=10, b=15;  **calc\_media(a, b);**  return 0;  } | #include <stdio.h>  #include <stdlib.h>  float calc\_media(int a, int b) {  float m;  m = (a+b)/2.0;  return(m);  }  int main(){  int a=10, b=15;  float media;  **media = calc\_media(a, b);**  printf("%.2f", media);    return 0;  } |

**Na primeira situação, a função calc\_media é chamada e não há atribuição do valor de retorno para nenhuma variável. Na segunda situação, a função calc-media retorna o valor da variável local m para a variável local média.**

## COMANDO RETURN

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

*return valor\_de\_retorno;* ou *return;*

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 compatível com o tipo de retorno declarado para a função.

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**. Abaixo estão dois exemplos de uso do **return**:

|  |  |
| --- | --- |
| #include <stdlib.h>  #include <stdio.h>  int Square (int a){  return (a\*a);  }  int main () {  int num;  printf ("Entre com um numero: ");  scanf ("%d",&num);  num=Square(num);  printf ("\n\nQuadrado = %d\n",num);  return 0;  } | #include <stdio.h>  #include <stdlib.h>  int EhMaior (int a){  if (a>=10) // Verifica se é maior ou igual a 10  **return 1;** // Retorna 1 se for igual o maior  else  **return 0;** // Retorna 0 se for menor que 10  }  int main (){  int num;  printf ("Entre com numero: ");  scanf ("%d",&num);  if (EhMaior(num))  printf ("\n\n Eh igual ou maior que 10.\n");  else  printf ("\n\n Eh menor que 10.\n");    return 0;  } |

É importante notar que, como as funções retornam valores, podemos aproveitá-los para fazer atribuições, ou mesmo para que estes valores participem de expressões. Mas *não* podemos fazer:

func(a,b)=x;    /\* Errado! \*/

No segundo exemplo vemos o uso de mais de um **return** em uma função.

Se uma função retorna um valor você *não precisa aproveitar* este valor. Se você não fizer nada com o valor de retorno de uma função ele será descartado. Por exemplo, a função **printf()** retorna um inteiro que nós nunca usamos para nada. Ele é descartado.

## FUNÇÕES QUE NÃO RECEBEM PARÂMETROS E NÃO RETORNARM VALORES – USO DO TIPO VOID

Agora vamos ver o único tipo da linguagem C que não detalhamos ainda: o **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. Podemos agora escrever o protótipo de uma função que não retorna nada:

*void nome\_da\_função (declaração\_de\_parâmetros);*

Numa função, como a acima, não temos valor de retorno na declaração [**return**](../../Sistemas%20de%20Informação/Linguagem%20C/Programacao%20C/Tutorial%20UFMG/aulas/c720.html). Aliás, neste caso, o comando [**return**](../../Sistemas%20de%20Informação/Linguagem%20C/Programacao%20C/Tutorial%20UFMG/aulas/c720.html) não é necessário na função.

Podemos, também, fazer funções que não têm parâmetros:

*tipo\_de\_retorno nome\_da\_função (void);*

 ou, ainda, que não tem parâmetros e não retornam nada:

*void nome\_da\_função (void);*

 Um exemplo de função que usa o tipo **void**:

#include <stdio.h>

#include <stdlib.h>

void Mensagem (void);

int main (){

Mensagem();

printf ("\tDiga de novo:\n");

Mensagem();

system("PAUSE");

return(0);

}

void Mensagem (void){

printf ("Ola! Eu estou vivo.\n");

}

Se quisermos que a função retorne algo, devemos usar a declaração [**return**](../../Sistemas%20de%20Informação/Linguagem%20C/Programacao%20C/Tutorial%20UFMG/aulas/c720.html). Se não quisermos, basta declarar a função como tendo tipo-de-retorno **void**. Devemos lembrar agora que a função **main()** *é* uma função e como tal devemos tratá-la. O compilador acha que a função **main()** deve retornar um inteiro. Isto pode ser interessante se quisermos que o sistema operacional receba um valor de retorno da função **main()**. Se assim o quisermos, devemos nos lembrar da seguinte convenção**: se o programa retornar zero, significa que ele terminou normalmente, e, se o programa retornar um valor diferente de zero, significa que o programa teve um término anormal**. Se não estivermos interessados neste tipo de informação, basta declarar a função main como retornando **void**.

 As duas funções **main()** abaixo são válidas:

main (void){

....

return 0;

}

void main (void){

....

}

A primeira forma é válida porque, como já vimos,  as funções em C têm, por padrão, retorno inteiro. Alguns compiladores reclamarão da segunda forma de main, dizendo que main sempre deve retornar um inteiro.

Mais um exemplo desse tipo de função (cuidado com o som):

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

void alarme(void) {

int f;

for (f=0; f<=1000; f+=20) {

Beep(f, 500); // frequência, tempo

}

}

int main() {

alarme();

system("PAUSE");

return(0);

}

Essa função é usada para emitir sons, variando a frequência.

## FUNÇÕES SEM PASSAGEM DE PARÂMETROS

## Acabamos de ver então um exemplo de uma função sem passagem de parâmetros e sem valor de retorno. Vamos ver uma aplicação dessa função:

## #include <stdio.h>

## #include <stdlib.h>

## #include <windows.h>

## #define SENHA 123

## void alarme(void);

## int main() {

## int s;

## printf("Informe a senha:");

## scanf("%d", &s);

## if (s != SENHA) {

## printf("\nSenha invalida!");

## alarme();

## }

## else printf("\nSenha ok!");

## return(0);

## }

## void alarme(void) {

## int f;

## for (f=200; f<=1000; f+=50) {

## Beep(f,100); // frequencia, tempo

## }

## }

Nesse exemplo, apenas chamamos a função **alarme** de dentro do programa principal, sem enviarmos argumentos para os parâmetros formais ou utilizarmos retornos.

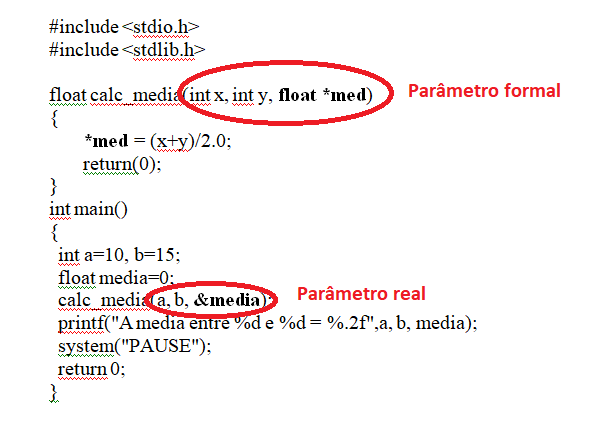
## FUNÇÕES COM PASSAGEM DE PARÂMETROS

As funções com passagem de parâmetros recebem parâmetros através da chamada das funções utilizando argumentos. A passagem de parâmetros pode ser feita por valor ou por referência.

**IMPORTANTE:**

* Na passagem de parâmetros ***por valor* não ocorre modificação** dos conteúdos dos argumentos.
* Na passagem de parâmetros ***por referência*** **ocorre modificação** dos conteúdos dos argumentos.

Os parâmetros que são “enviados” para a função são chamados de **“parâmetros reais”.**



## FUNÇÕES COM PASSAGEM DE PARÂMETROS POR VALOR

## Por default, os argumentos em C são passados por valor, ou seja, quando uma função é chamada, um novo espaço de memória é alocado para cada um de seus parâmetros e os valores dos argumentos correspondentes são copiados nesses espaços. A consequência disso é que nenhuma alteração feita pela função e seus parâmetros pode afetar os valores dos argumentos que lhe foram passados.

## Uma das vantagens da passagem de parâmetros por valor é que as funções ficam impedidas de acessar variáveis declaradas em outras funções.

## Lembrar: na passagem de valores são passados para a função apenas os valores dos argumentos para os parâmetros e não os próprios argumentos, ou seja, os valores não poderão ser modificados.

## #include <stdio.h>

## #include <stdlib.h>

## float calc\_media(int x, int y){

## float m;

## m = (x+y)/2.0;

## x = x\*100;

## return(m);

## }

## int main() {

## int a=10, b=15;

## float media=0;

## media = calc\_media(a, b);

## printf("A media entre %d e %d = %.2f",a, b, media);

## system("PAUSE");

## return 0;

## }

## Nesse programa, os conteúdos das variáveis a e b foram passados para os parâmetros formais x e y. Assim, x e y terão os valores de a e b respectivamente, quando a função calc\_media for chamada.

## É importante lembrar que os parâmetros formais devem ser do mesmo tipo dos argumentos e estar na mesma ordem desejada.

## IMPORTANTE: Note que o valor de x foi alterado dentro da função, mas a alteração não se reflete no conteúdo da variável a.

## FUNÇÕES COM PASSAGEM DE PARÂMETROS POR REFERÊNCIA

## Como já foi comentado antes, uma das vantagens da passagem de parâmetros por valor é que as funções ficam impedidas de acessar variáveis declaradas em outras funções. Entretanto, algumas vezes deseja-se que isso seja possível. Para tal será utilizada a passagem de parâmetros por referência.

## Lembrar: na passagem de valores por referência os endereços dos argumentos são passados para a função, ou seja, os valores poderão sim ser modificados.

## 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. Ainda, quando quisermos chamar a função, teremos de lembrar-nos de colocar um & na frente das variáveis (argumentos) que estivermos passando para a função.

Espere um momento... será que nós já não vimos esta estória de chamar uma função com as variáveis precedidas de **&**? Já! É assim que nós chamamos a função **scanf()**. Mas por quê? Vamos pensar um pouco. A função **scanf()** usa chamada por referência porque ela precisa alterar as variáveis que passamos para ela! Não é para isto mesmo que ela é feita? Ela lê variáveis para nós e, portanto, precisa alterar seus valores. Por isto passamos para a função o **endereço da variável** a ser modificada!

Vejam o mesmo exemplo já utilizado antes, agora com passagem de parâmetros por referência.

#include <stdio.h>

#include <stdlib.h>

void calc\_media(int x, int y, **float \*med**){

**\*med** = (x+y)/2.0;

}

int main(){

int a=10, b=15;

float media=0;

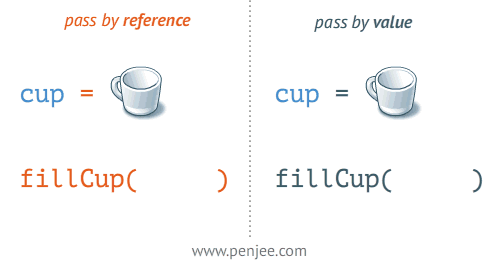
calc\_media(a, b, **&media**);

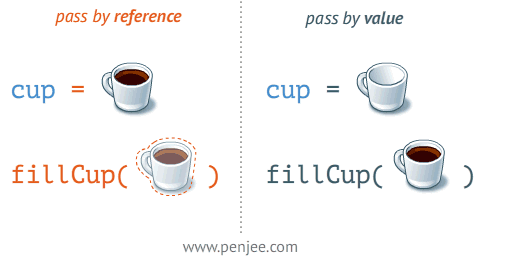
printf("A media entre %d e %d = %.2f",a, b, media);

return 0;

}

Agora, **o endereço da variável media** é passado como argumento para o ponteiro **\*med** na função. Assim, as modificações em **\*med** serão refletidas no conteúdo da variável **media**.





## VETORES COMO PARÂMETROS DE FUNÇÕES

## Os vetores são sempre definidos como parâmetro por referência.

## void func(int vetor[10])

## É possível não definir a dimensão do vetor, mas deve-se tomar cuidado para não extrapolar a dimensão permitida.

## func(v);

## 

## void func(int vetor[])

## É comum passar o tamanho do vetor como parâmetro para a função

## void func(int vetor[], int tamanho)

## O vetor não pode ser utilizado como retorno de uma função.

## Abaixo está um exemplo de programa com uma função para imprimir o vetor:

## #include <stdio.h>

## #include <stdlib.h>

## #include <time.h>

## void imprime(int vetor[], int tamanho);

## int main() {

## int abc;

## int vet[100], cont;

## srand(time(NULL));

## for (cont=0; cont<100; cont++)

## vet[cont] = rand()%20;

## imprime(vet, 100); // diferente de char imprime(vet[10], 100);

## return(0);

## }

## void imprime(int vetor[], int tamanho) {

## int cont;

## printf("Vetor gerado\n");

## for (cont=0; cont<tamanho; cont++)

## printf("%5d", vetor[cont]);

## printf("\n\n");

## }

## MATRIZES COMO PARÂMETROS DE FUNÇÕES

## Como os vetores, as matrizes também são definidas como parâmetro por referência

## void func(int mat[10][10])

## void func(int mat[10][10], int linhas, int colunas)

## Só é permitido não definir a primeira dimensão da matriz (vetor), mas as demais devem ser definidas.

## Uma Mariz também não pode ser utilizada como resultado de uma função.

## #include <stdio.h>

## #include <stdlib.h>

## #include <time.h>

## #define TAM 5

## 

## void imprime(int matriz[5][5], int a, int b);

## int main()

## {

## int mat[TAM][TAM],l,c;

## srand(time(NULL));

## for(l=0; l<5; l++)

## for(c=0; c<5; c++)

## mat[l][c] = rand()%20;

## imprime(mat, TAM, TAM);

## 

## system("PAUSE");

## return(0);

## }

## void imprime(int matriz[5][5], int a, int b) {

## int lin, col;

## printf("Matriz gerada \n");

## for(lin=0; lin<a; lin++){

## for(col=0; col<b; col++)

## //printf("%5d", matriz[lin][col]);

## matriz[lin][col] = 10;

## printf("\n"); }

## printf("\n\n");

## }

## ARQUIVOS-CABEÇALHOS

Arquivos-cabeçalhos são aqueles que temos mandado o compilador incluir no início de nossos exemplos e que sempre terminam em **.h**. A extensão **.h** vem de **header** (cabeçalho em inglês). Já vimos exemplos como **stdio.h**, **conio.h**, **string.h**. Estes arquivos, na verdade, não possuem os códigos completos das funções. Eles só contêm *protótipos* de funções. É o que basta. O compilador lê estes protótipos e, baseado nas informações lá contidas, gera o código correto. O corpo das funções cujos protótipos estão no arquivo-cabeçalho, no caso das funções do próprio C, já estão compiladas e normalmente são incluídas no programa no instante da "linkagem". Este é o instante em que todas as referências a funções cujos códigos não estão nos nossos arquivos fontes são resolvidas, buscando este código nos arquivos de bibliotecas.

Se você criar algumas funções que queira aproveitar em vários programas futuros, ou módulos de programas, você pode escrever arquivos-cabeçalhos e incluí-los também.

Suponha que a função 'int EPar(int a)', seja importante em vários programas, e desejemos declará-la num módulo separado. No arquivo de cabeçalho chamado, por exemplo, de 'funcao.h' teremos a seguinte declaração:

#include "funcao.c"

int EPar(int a);

O código da função será escrito num arquivo à parte. Vamos chamá-lo de 'funcao.c'. Neste arquivo teremos a definição da função:

int EPar (int a){

if (a%2)             /\* Verifica se a e divisível por dois \*/

        return 0;

else

        return 1;

}

Por fim, no arquivo do programa principal teremos o programa principal. Vamos chamar este arquivo aqui de 'princip.c'.

#include <stdio.h>

#include "funcao.h"

void main ()

{

int num;

printf ("Entre com numero: ");

scanf ("%d",&num);

if (EPar(num))

        printf ("\n\nO numero e par.\n");

else

        printf ("\n\nO numero e impar.\n");

}

Será preciso compilar primeiro o arquivo funcao.c e ao compilar o arquivo princip.c será necessário incluir funcao.o nas opções do projeto.

## EXERCÍCIOS

* 1. Estude o seguinte programa e aponte o valor de cada variável sempre que solicitado:

#include <stdio.h>

#include <stdlib.h>

int num;

int func(int a, int b){

a = (a+b)/2; // Qual e o valor de a após a atribuição?

num -= a; // qual é o valor de num ?

return a; // que valor retorna para o programa ?

}

main() {

int first = 0, sec = 50;

num = 100;

num += func(first, sec); // Qual e o valor de num, first e sec

// antes e depois da atribuicao?

printf("\n\nConfira! num = %d\tfirst = %d\tsec = %d\n",num, first, sec);

system("PAUSE");

return(0);

}

* 1. Construir um programa que solicita ao usuário um número inteiro. Em seguida, uma função deve ser chamada para contar quantos números pares existem entre 0 e o número digitado. A função deve contar a quantidade de retornar o resultado para o programa principal. Construir a função com passagem de parâmetros por valor e com retorno.
  2. Escrever um programa que lê 2 números inteiros e em seguida chama a função EDivisivel. A função 'EDivisivel(int a, int b)' deverá retornar 1 se o resto da divisão de a por b for zero, ou seja, se a for divisível por b. Caso contrário, a função deverá retornar zero.
  3. Construir um programa que lê caracteres, em seguida chama uma função que testa se cada um dos caracteres é a letra A. Usar passagem de parâmetros por valor. Se o caractere for A, a função deve retornar 1, caso contrário, deve retornar 0. Devem ser lidos e testados, um a um, 5 caracteres.
  4. Escreva um programa com uma função chamada Troca. Inicialmente devem ser atribuídos os valores 100 e 200 para as variáveis locais num1 e num2, respectivamente. Em seguida, a função Troca deve ser chamada. A função deve fazer a troca dos valores, utilizando passagem de parâmetros por referência.
  5. Considerando o programa abaixo, que mensagem será exibida na tela quando ele for executado?

|  |
| --- |
| #include <stdlib.h>  int Mostra1(int n);  int Mostra2(int \*n);  int main() {  int num = 10;  Mostra2(&num);  printf("%d\n", num);  system("PAUSE");  return(0);  }  int Mostra1(int n){  n = 100;  return(0);  }  int Mostra2(int \*n){  int x;  \*n = 200;  x = \*n;  Mostra1(x);  return(0);  } |

* 1. Construir um programa que leia 2 números inteiros e positivos (n1 e n2). O programa deve ter uma função denominada MaiorMenor que coloque o menor valor na variável n1 e o maior em n2. Deve existir ainda outra função denominada Mostra que mostre os números inteiros existentes entre n1 e n2, inclusive. Ambas deverão ser chamadas pelo programa principal, com passagem de parâmetros por valor ou por referência.
  2. Faça um programa com uma função Calculando. A função deve ter 4 parâmetros inteiros. Esta função deverá calcular a soma dos 3 primeiros parâmetros, retornando para o programa principal o resultado no quarto parâmetro. Utilize passagem de parâmetros por valor e por referência, conforme as necessidades. A função deve retornar ainda o produto dos quatro parâmetros. O programa principal deve mostrar a soma e o produto calculados pela função.
  3. Construir um programa que lê valores inteiros para um vetor de 10 posições e em seguida chama uma função para mostrar os valores vetor. Construir uma função com passagem de parâmetros por valor. Note que para chamar a função devemos usar NomeDaFunção(NomeDoVetor) e para declarar o parâmetro devemos fazer NomeDaFunção(tipo NomeDoVetor[]).
  4. Construir um programa que sorteie valores para um vetor de 5 posições. Logo depois chame a função Mostra para mostrar o vetor sem ordenar. Em seguida deve ser chamada a função Bubble para ordenar o vetor. Depois deve ser chamada a função Mostra que mostre o vetor já ordenado. Utilize funções com passagem de parâmetros por valor. Lembre que com vetores não será necessário utilizar ponteiros, pois ao passarmos um vetor para uma função, na realidade estamos passando um ponteiro. Neste ponteiro é armazenado o endereço 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 dos elementos do vetor dentro da função.

void bubble\_sort1 (int vetor[], int n) {

int k, j, aux;

for (k = 1; k < n; k++) {

printf("\n[%d] ", k);

for (j = 0; j < n - 1; j++) {

printf("%d, ", j);

if (vetor[j] > vetor[j + 1]) {

aux = vetor[j];

vetor[j] = vetor[j + 1];

vetor[j + 1] = aux;

}

}

}

}

void bubble\_sort2 (int vetor[], int n) {

int k, j, aux;

for (k = 1; k < n; k++) {

printf("\n[%d] ", k);

for (j = 0; j < n - k; j++) {

printf("%d, ", j);

if (vetor[j] > vetor[j + 1]) {

aux = vetor[j];

vetor[j] = vetor[j + 1];

vetor[j + 1] = aux;

}

}

}

}

void bubble\_sort3 (int vetor[], int n) {

int k, j, aux;

for (k = n - 1; k > 0; k--) {

printf("\n[%d] ", k);

for (j = 0; j < k; j++) {

printf("%d, ", j);

if (vetor[j] > vetor[j + 1]) {

aux = vetor[j];

vetor[j] = vetor[j + 1];

vetor[j + 1] = aux;

}

}

}

}

void bubble\_sort4 (int vetor[], int n) {

int k =0, j, aux;

int trocas = 1;

while(trocas){

trocas = 0;

k++;

printf("\n[%d] ", k);

for (j = 0; j < n; j++) {

printf("%d, ", j);

if (vetor[j] > vetor[j + 1]) {

aux = vetor[j];

vetor[j] = vetor[j + 1];

vetor[j + 1] = aux;

trocas = 1;

}

}

}

}