Funções e Estrutura de Blocos

Funções devolvem em geral um valor calculado pela mesma. O próprio programa principal é uma função. Em vez de `main ()` podemos escrever `int main()`, significando isto que o programa principal devolve um valor do tipo `int`.

Vamos agora a um exemplo.

Dado `n` e `p` inteiros, `n,p >= 0`, calcular as combinações de `n` elementos `p` a `p`, isto é `n!/(p! * (n-p)!)`.

Se existisse uma função `fat(x)`, pré-definida que calculasse o fatorial de um inteiro `x >= 0`, o programa ficaria assim:

```
#include <stdio.h>
/* Dado n e p inteiros, n,p >= 0, calcular as combinações de n elementos
   p a p, isto é, n!/(p! * (n-p)!). */

int main() {
 int n, p; /* dados */

 /* ler n e p com consistência */
 while (0 == 0) { /* repita sempre */
    printf("\ndigite os valores de n e p separados por branco(s):");
    scanf("%d%d", &n, &p);
      if (n >= 0 && p >= 0) break;
      else printf("\n***n ou p digitados errados");
 }

 /* calcular, usando a função fat */
 printf("\n*** combinacoes de %5d -  %5d a %5d = %10d", n, p, p,
          fat(n)/(fat(p)*fat(n-p)));
}
```

Como não existe, é necessário definir-se a função `fat` neste programa. Veja abaixo como isso é feito.

```
#include <stdio.h>
/* Dado n e p inteiros, n,p >= 0, calcular as combinações de n elementos
   p a p, isto é, n!/(p! * (n-p)!). */

int fat(int k) {
   /* função que devolve o fatorial de k */
   int i, /* contador */
       f; /* para o cálculo do fatorial */

    f = 1;
    for (i = 1; i <= k; i++) f = f * i;
    return(f);
}

int main() {
 int n, p; /* dados */

 /* ler n e p com consistência */
```

```
 while (0 == 0) { /* repita sempre */
    printf("\ndigite os valores de n e p separados por branco(s):");
    scanf("%d%d", &n, &p);
      if (n >= 0 && p >= 0) break;
      else printf("\n***n ou p digitados errados");
 }

 /* calcular, usando a função fat */
 printf("\n*** combinacoes de %5d -  %5d a %5d = %10d", n, p, p,
          fat(n)/(fat(p)*fat(n-p)));
}
```

Com isso, criamos uma nova função chamada `fat`.

Algumas observações:

a) `fat` só pode ser usada no programa que foi definida.
b) Se for necessário usar a função `fat` em outros programas, pode-se deixá-la num arquivo do tipo `.h` e dar `#include`.
c) O comando `return <valor>` retorna da função, isto é, retorna para o programa que chamou a função, devolvendo `valor`, como o resultado da função.
d) Não é obrigatório o comando `return` numa função. Se não houver, retorna da função no final com resultado indefinido.
e) `return(x)` e `return x` é a mesma coisa. Não são necessários os parêntesis.
f) Os argumentos da função precisam ser declarados, pois precisam ser de algum tipo específico.
g) A própria função tem um tipo. Se não houver necessidade de um tipo específico, ou seja, a função não precisa devolver um valor, usar o tipo vazio `void`.
h) O formato geral de declaração de uma função é:

```
tipo nome(declaração dos parâmetros formais)
  {declarações e comandos}
```

Exemplos de declaração de funções:

```
int f1(int x, float y)
    {:
     :
     return z
    }

void f2(void);
    {:
     :
    }
```

Função `abs(x)` que calcula `|x|`. Supondo que não seja uma função pré-definida.

```
#include <stdio.h>
#include <math.h>
/* Função abs(x) que calcula |x|. Supondo que abs não seja uma função
   pré-definida. */
```

```
double abs(double x) {
   /* função que devolve o |x|
   if (x >= 0) return x;
   else return -x;
}

/* Exemplo de programa principal que usa abs.
   Dados dois reais calcule a raiz quadrada de seu produto em módulo */

int main() {
 double a, b; /* dados */

 /* ler a e b */
 printf("\ndigite os valores de a e b separados por branco(s):");
 scanf("%lf%lf", &a, &b);

 /* calcular e imprimir */
 printf("\n*** resultado = %15.5lf", sqrt(abs(a*b)));
}
```

Função `maior(x,y)` que calcula o maior entre as duas várias do tipo `double x e y.`

```
#include <stdio.h>
/* Função maior(x,y) que calcula o maior entre dois double x e y. */

double maior(double x, double y) {
   /* função que devolve o maior entre x e y */
    if (x > y) return x;
    else return y;
}

/* Exemplo de programa principal que usa maior.
   Dados 3 números imprimir o maior entre eles. */

int main() {
 double a, b, c; /* dados */

 /* ler a b e c *
 printf("\ndigite os valores de a, b e c separados por branco(s):");
 scanf("%lf%lf%lf", &a, &b, &c);

 /* calcular e imprimir */
 printf("\n*** maior = %15.5lf", maior(a, maior(b, c)));
}
```

Outra versão da função `maior`:

```
double maior(double x, double y) {
   /* função que devolve o maior entre x e y */
   if (x > y) return x;
   return y;
}
```

Função `impmaior(x,y)` que imprime o maior valor entre dois `double x` e `y`

```
#include <stdio.h>
/* Função impmaior(x,y) que imprime o maior entre dois double x e y. */
```

```
void impmaior(double x, double y) {
   /* imprime o maior entre x e y */
   if (x > y) printf("\n**** o maior e:%15.5lf", x);
   else printf("\n**** o maior e:%15.5lf", y);
}

/* Exemplo de programa principal que usa impmaior.
   Dados 2 números imprimir o maior entre eles. */

int main() {
 double a, b; /* dados */

 /* ler a e b */
 printf("\ndigite os valores de a e b separados por branco(s):");
 scanf("%lf%lf", &a, &b);

 /* imprimir */
 impmaior(a, b);

}
```

Observe que a função `impmaior` acima, não devolve nada, por isso seu tipo é `void`. Além disso, não precisa de `return`.

Uma outra versão:

```
void impmaior(double x, double y) {
   /* imprime o maior entre x e y */
   if (x > y) {printf("\n**** o maior e:%15.5lf", x); return;}
   printf("\n**** o maior e:%15.5lf", y);
}
```

Exercícios

1) Escreva uma função que dadas as notas de prova `p1` e `p2` e as notas dos exercícios-programa `ep1, ep2 e ep3` de um aluno, devolve o valor da média final deste aluno. A média final do aluno é dada por `(3p+ep)/4`, onde `p = (p1+2p2)/3` e `ep = (ep1+2ep2+3ep3)/6.`

2) Idem, acrescentando se `p<5` ou `ep<6` a média final é o mínimo entre `4.5, p` e `ep.`

3) Idem, acrescentando os parâmetros ps = prova substitutiva e eps = ep substitutivo. Se ps >= 0, ps substitui a menor entre p1 e p2, com o respectivo peso. Se eps >= 0, eps substitui a menor entre ep1, ep2 e ep3.

A função seno de x, pode ser calculada pela fórmula de Taylor da seguinte forma:

$\text{sen}(x) = x/1! - x^3/3! + x^5/5! - \ldots + (-1)^k \cdot x^{2k+1}/(2k+1)! + \ldots$

A série é infinita, mas seus termos, em módulo, são decrescentes. Portanto, para se obter uma boa aproximação de sen (x) , basta calcular a soma, até que termo corrente em módulo, seja menor que um número epslon bem pequeno, ou seja bem próximo de zero.

5) Escreva uma função seno(x,eps), `double seno(double x, double eps)`, que calcula o seno de x pela fórmula e critério acima.

Da mesma forma o cosseno pode ser calculado por:

$\cos(x) = 1 - x^2/2! + x^4/4! - x^6/6! + \ldots + (-1)^k \cdot x^{2k}/(2k)! + \ldots$

6) Idem à função cosseno(x,eps), `double cosseno(double x, double eps)`

A função arco tangente de x pode ser calculada por:

$\arctan(x) = x - x^3/3 + x^5/5 - x^7/7 + \ldots + (-1)^{2k-1} \cdot x^{2k-1}/(2k-1)$

7) idem a função arctan (x, eps), `double arctan(double x, double eps)`

Estrutura de Blocos

Considere o programa abaixo e suas funções:

```
int f1(int i) {
   int k, j;
   :
   :
   i = k + j;
}

int f2(int i, int j) {
   int k;
   :
   :
   i = j + k;
}

int f3(int j, int k) {
   :
   :
   i = j + k;
}

int main () {
   int i, j, k;
   :
   :
   if (i == j) {
       int k;
       :
       :
       i = k + j;
   }
   i = j + k;
   f1(i);
   f2(i + j, i + k);
   f3(i * j, k);
```

```
    :
}
```

Em qualquer bloco de comandos, podem-se fazer declarações de novas variáveis. Inclusive em blocos internos ao programa principal ou às funções, isso pode ocorrer. Essa declaração vale do início ao fim do bloco, isto é, enquanto comandos do bloco estiverem sendo executados, a variável declarada existe. Quando termina a execução dos comandos do bloco, as variáveis declaradas nele deixam de existir.

Observe ainda, que no exemplo acima existem variáveis declaradas com o mesmo nome. Como cada variável só existe dentro do bloco no qual foi declarada, não existe inconsistência.

No entanto, não é uma boa prática de programação se usar variáveis com mesmo nome, com exceção de contadores, i, j, etc. que claramente tem sempre o mesmo tipo de uso.

Vamos identificar as variáveis diferentes no programa acima.

```
int f1(int i1) {
    int k1, j1;
    :
    :
    i1 = k1 + j1;
}

int f2(int i2, int j2) {
    int k2;
    :
    :
    i2 = j2 + k2;
}

int f3(int j3, int k3) {
    :
    :
    i? = j3 + k3;  // erro de sintaxe. não existe i declarado neste bloco
}

main () {
    int i3, j4, k4;
    :
    :
    if (i3 == j4) {
        int k5;
        :
        :
        i3 = k5 + j4;
    }
    i3 = j4 + k4;
    f1(i3);
    f2(i3 + j4, i3 + k4);
    f3(i3 * j4, k4);
    :
}
```

O que será impresso pelo programa abaixo?

```
#include <stdio.h>
double func (int x) {
  int i, j;
  i = 0;
  j = 0;
  x = x + 1;
  printf("\nvalor de x dentro da funcao = %5d", x);
}

/* program principal */
int main () {
 int i, j;
 i = 10;
 j = 20;
 func(i) ; printf("\nvalor de i apos a chamada da funcao = %5d", i);
 func(j) ; printf("\nvalor de j apos a chamada da funcao = %5d", j);
 func(i+j) ; printf("\nvalor de i+j apos a chamada da funcao = %5d", i + j);
}
```

```
valor de x dentro da funcao =    11
valor de i apos a chamada da funcao =    10
valor de x dentro da funcao =    21
valor de j apos a chamada da funcao =    20
valor de x dentro da funcao =    31
valor de i+j apos a chamada da funcao =    30
```

Importante:

a) Os parâmetros são passados sempre por valor. Em outras palavras, o que é passado como parâmetro é o valor da variável ou da expressão. Não importa qual variável ou expressão.
b) Alterar um parâmetro dentro da função, não altera a variável no programa principal. É como se o parâmetro da função fosse uma nova variável que recebe o valor passado como parâmetro no início da função.

Variáveis Locais e Variáveis Globais

Já vimos que as variáveis declaradas dentro de um bloco, existem apenas dentro deste bloco, ou seja são locais a este bloco, não sendo conhecidas fora do mesmo.

Variáveis que são declaradas dentro de blocos que possuem outros blocos internos, são ditas globais aos blocos internos. Veja o exemplo abaixo e diga o que será impresso.

```
#include <stdio.h>
/* program principal */
int main () {
 /* bloco principal */
 int i, j;
 i = 10;
 j = 20;

 if (i < j) {
     /* bloco interno */
     int k;
     k = i + j;
```

```
     printf("\nvalor de k dentro do bloco interno = %5d", i);
     i = k;
     j = k + 10;
 }

 printf("\nvalor de i no bloco principal apos o bloco interno = %5d", i);
 printf("\nvalor de j no bloco principal apos o bloco interno = %5d", j);
}
```

```
valor de k dentro do bloco interno =    10
valor de i no bloco principal apos o bloco interno =    30
valor de j no bloco principal apos o bloco interno =    40
```

As variáveis i e j são locais ao bloco principal e globais ao bloco interno. A variável k é local ao bloco interno e desconhecida no bloco principal.
No caso de variáveis globais à funções e ao programa principal, existe uma forma de declará-las. Veja o exemplo abaixo. As variáveis a e b, declaradas fora do programa principal e fora das funções func1 e func2, são globais a esses três e como tal são conhecidas dentro dos mesmos.

O que será impresso pelo programa abaixo?

```
#include <stdio.h>
int a;
double b;
double func1 (int x) {
  x = x + a;
  printf("\nvalor de x dentro da funcao func1 = %5d", x);
  return b;
}

int func2 (double x) {
  x = x + b;
  printf("\nvalor de x dentro da funcao func2 = %10.5lf", x);
  return a;
}

/* program principal */
int main () {
 a = 20;
 b = 3.1416;
 printf("\nvalor de func1 (0) = %10.5lf", func1 (0));
 printf("\nvalor de func2 (0.0) = %5d", func2 (0.0));
}
```

```
valor de x dentro da funcao func1 =    20
valor de func1 (0) =    3.14160
valor de x dentro da funcao func2 =    3.14160
valor de func2 (0.0) =    20
```

Váriáveis Dinâmicas e Estáticas

Conforme já vimos acima, uma variável local a um bloco, existe apenas dentro deste bloco. Quando termina a execução deste bloco, a variável desaparece. Se esse bloco é novamente executado, outra nova variável é criada, possivelmente ocupando uma outra posição de memória. Dessa forma, diz-se que as variáveis são dinâmicas, pois são criadas e consumidas durante a execução do programa.

Veja o exemplo abaixo e o que será impresso

```
#include <stdio.h>

double func (int x) {
  /* declaração de variável dinâmica. O valor da variável
     na chamada anterior não permanece nas chamadas posteriores */
  int s = 0,
      v = 0;
  v++;
  printf("\n\n%2d.a chamada - valor de s no inicio = %5d", v, s);
  s = s + x;
  printf("\n%2d.a chamada - valor de s no final  = %5d", v, s);
}

/* program principal */
int main () {
 func (10);
 func (20);
 func (30);
 func (40);
}
```

```
 1.a chamada - valor de s no inicio =     0
 1.a chamada - valor de s no final  =    10

 1.a chamada - valor de s no inicio =     0
 1.a chamada - valor de s no final  =    20

 1.a chamada - valor de s no inicio =     0
 1.a chamada - valor de s no final  =    30

 1.a chamada - valor de s no inicio =     0
 1.a chamada - valor de s no final  =    40
```

Existe uma forma de criar uma variável no início do programa e fazer com que ela mantenha os valores a ela atribuído, independente do bloco onde a mesma foi declarada. É feito com o atributo **`static`**.

Veja o exemplo abaixo e o que será impresso.

```
#include <stdio.h>

double func (int x) {
  /* Declaração de 2 variáveis locais porém estáticas.
     Permanecem com o valor da chamada anterior. */
  static int s = 0,
             v = 0;
  v++;
  printf("\n\n%2d.a chamada - valor de s no inicio = %5d", v, s);
  s = s + x;
  printf("\n%2d.a chamada - valor de s no final  = %5d", v, s);
}

/* Program principal */
int main () {
 func (10);
```

```
 func (20);
 func (30);
 func (40);
}
```

```
 1.a chamada - valor de s no inicio =     0
 1.a chamada - valor de s no final  =    10

 2.a chamada - valor de s no inicio =    10
 2.a chamada - valor de s no final  =    30

 3.a chamada - valor de s no inicio =    30
 3.a chamada - valor de s no final  =    60

 4.a chamada - valor de s no inicio =    60
 4.a chamada - valor de s no final  =   100
```

Variáveis em Registradores

Uma variável que é muito utilizada num programa ou num trecho de programa, pode ser declarada de forma que a mesma seja alocada a um registrador em vez de uma determinada posição de memória. Isso é feito com a declaração register. Exemplos:

```
register int x;
register char a;
register unsigned short k;
```

O atributo register só pode ser usado para variáveis dinâmicas e aos parâmetros formais de uma função, os quais também são dinâmicos. Exemplos:

```
int f(register int p1, register unsigned char a) {
   register int k;
   :
   :
}
main() {
   register int j;
   :
   :
      {register char aa;
       :
       :
      }
}
```

Como a quantidade de registradores disponíveis varia de processador para processador e em geral são poucos os registradores disponíveis, há restrições para a quantidade de variáveis register existentes num programa. Dessa forma essa, o atributo **`register`** pode ser ignorado pelo compilador devido a essas limitações.

Parâmetros de Retorno de Funções

Já vimos que uma função devolve, ou pode devolver um valor que é o resultado calculado pela função. E quando é necessário que mais de um valor seja calculado e devolvido pela função??? Por exemplo,

suponha uma função que receba a, b e c, calcule e devolva as raízes da equação do 2. grau ax2 + bx + c = 0. Neste caso, 2 valores precisam ser devolvidos.

Já vimos também, que a alteração de um parâmetro dentro da função não altera o valor da variável que foi passada como parâmetro como no exemplo abaixo:

```
#include <stdio.h>
double f (int x) {
  x = x + 20;
}
int main () {
   int a;
    :
    :
    f (a);
    :
}
```

A variável a não é alterada pela função f.

E se for necessário que haja a alteração da variável passada como parâmetro?

Existe uma maneira de resolver os dois problemas acima. E o jeito é a possibilidade de termos parâmetros de retorno. Os parâmetros de uma função podem ser classificadas em parâmetros de entrada (aqueles que usamos até o momento e cujo valor são necessários para a execução da função) e os parâmetros de saída (aqueles que são calculados e devolvidos ao programa ou função que chamou).

Para que um parâmetro seja de saída, o mesmo deve ser passado para a função como um endereço ou uma posição de memória.

Endereços e Ponteiros

Se **x** é uma variável, então **&x** designa o endereço de **x**, isto é, a localização de **x** na memória do computador. O acesso indireto à **x**, isto é, através de seu endereço, é indicado por ***x**. ***x** também é chamado de ponteiro para x.

Veja o exemplo abaixo. O que será impresso?

```
#include <stdio.h>
double func (int *x) {
  /* altera a variável que foi passada como parâmetro */
  *x = *x + 10;
}

/* program principal */
int main () {
 int a = 20; /* declara variável a e inicia com 20 */
 printf("\nvalor de a antes de chamar func = %5d", a);
 /* chama func passando o endereço de a como parâmetro */
 func(&a);
 printf("\nvalor de a apos chamada de func = %5d", a);
}
```

```
valor de a antes de chamar func =    20
valor de a apos chamada de func =    30
```

Veja agora outro exemplo e o que será impresso. No exemplo abaixo imprimimos também o endereço passado como parâmetro:

```
#include <stdio.h>

double imp_valor_end (int vez, int *x) {
  /* imprime o valor e o endereço de x */
  printf("\n\n\n********** imprime na %3d-esima vez ************", vez);
  printf("\n** o valor do parametro *x = %5d", *x);
  printf("\n** o endereco do parametro x = %p", x);
}

/* program principal */
int main () {
 int a = 20; /* declara variável a e inicia com 20 */
 int *enda;  /* conterá o endereço de a */

 imp_valor_end (1, &a);

 enda = &a;
 imp_valor_end (2, enda);

 /* altera o valor de a indiretamente. Note que enda já contém &a */
 *enda = 30;
 imp_valor_end (3, enda);
}
```

```
********** imprime na   1-esima vez ************
** o valor do parametro *x =    20
** o endereco do parametro x = 0063FDE4


********** imprime na   2-esima vez ************
** o valor do parametro *x =    20
** o endereco do parametro x = 0063FDE4


********** imprime na   3-esima vez ************
** o valor do parametro *x =    30
** o endereco do parametro x = 0063FDE4
```

Agora vamos resolver alguns problemas:

Escreva uma função que recebe 3 valores reais a, b e c (parâmetros de entrada) e devolve as raízes da equação (parâmetros de saída) $ax^2 + bx + c = 0$

Como existem vários casos particulares, vamos fazer com com o valor devolvido pela função nos dê a informação sobre isso. Assim os valores possíveis são:

devolve `1` – quando há duas raízes diferentes
devolve `0` – quando há duas raízes iguais, ou seja delta é zero

devolve −1 – quando não há raízes reais
devolve −2 – quando a = 0 e b != 0, ou seja, é equação do primeiro grau
devolve −3 – quando a = 0 e b = 0, ou seja não é equação

```
#include <stdio.h>
#include <math.h>
/* Função que recebe 3 valores reais a, b e c (parâmetros de entrada).
   Devolve as raízes da equação (parâmetros de saída) ax2 + bx + c = 0.
   Devolve como valor da função:
      1 - quando há duas raízes diferentes
      0 - quando há duas raízes iguais, ou seja delta é zero
     -1 - quando não há raízes reais
     -2 - quando a = 0 e b != 0,ou seja, é equação do primeiro grau
     -3 - quando a = 0 e b = 0, ou seja não é equação
*/

int equacao_2grau (double a, double b, double c, double *x1, double *x2) {
   double delta;

   /* verifica se a = b = 0. Neste caso não é equação. */
   if (a == 0.0 && b == 0.0) return -3;
   /* se apenas a = 0, devolve apenas a raiz x1, pois é equação do 1.grau */
   if (a == 0.0) {*x1 = -c / b; return -2;}
   /* calcule o valor do discriminante */
   delta = b*b - 4*a*c;
   /* verifica se delta < 0 e neste caso não há raizes reais */
   if (delta < 0) return -1;
   /* verifica se delta = 0 e neste caso as duas raízes são iguais. */
   if (delta == 0) {*x1 = *x2 = -b / (2*a); return 0;}
   /* neste caso as duas raízes são diferentes. */
   *x1 = (-b + sqrt(delta))/(2*a);
   *x2 = (-b - sqrt(delta))/(2*a);
   return 1;
}
```

Um exemplo de programa principal que usa a função equacao_2grau é mostrado abaixo.

```
int main() {
 double c1, c2, c3,    /* coeficientes dados */
        raiz1, raiz2;  /* raízes calculadas */

 /* entre com c1, c2 e c3 */
 printf("digite o valor de c1:");
 scanf("%lf", &c1);

 printf("digite o valor de c2:");
 scanf("%lf", &c2);

 printf("digite o valor de c3:");
 scanf("%lf", &c3);

 /* imprime resultado dependendo do caso */
 switch ( equacao_2grau(c1, c2, c3, &raiz1, &raiz2) ) {
     case  1: printf("\n2 raizes reais ** r1 = %15.5lf ** r2 = %15.5lf",
              raiz1, raiz2);
              break;
     case  0: printf("\n2 raizes reais iguais ** r1 = r2 = %15.5lf",
```

```
                raiz1);
                break;
      case -1: printf("\n ** nao ha raizes reais");
                break;
      case -2: printf("\n equacao do 1. grau ** raiz = %15.5lf", raiz1);
                break;
      case -3: printf("\n nao e equacao");
                break;
    }
}
```

Neste programa aparece um comando que ainda não vimos. O comando switch. É usado, quando dependendo do valor de uma variável inteira, o programa deve tomar alguma ação.

O formato geral do comando switch é:

```
switch (expressão inteira)
  case expressão constante inteira 1 : comandos 1;
                                       break;
  case expressão constante inteira 2 : comandos 2;
                                       break;
  :
  :
  :
  case expressão constante inteira n : comandos n;
                                       break;
  default: comandos n+1;
           break;
```

A condição default é opcional no comando switch. São executados os comandos i, se o valor da expressão inteira for i. Caso esse valor não seja igual a nenhum valor entre 1 e n, são executados os comandos n+1.

Outro exemplo de função com parâmetros de retorno

Escreva uma função que receba 2 variáveis reais e permute o valor destas variáveis.

Neste caso os dois parâmetros são de entrada e saída também.

```
// Função que receba 2 variáveis reais e permuta o valor destas variáveis. */

int troca (double *a, double *b) {
   double aux;
   aux = *a;
   *a = *b;
   *b = aux;
}
```

Um exemplo de programa principal que usa a função troca:

```
int main() {
 double x = 2.78,
        y = 3.14,
        *xx = &x,
```

```
        *yy = &y;

 /* x e y antes e depois da troca */
 printf("\n\n\n************** primeiro exemplo ******************");
 printf("\n* antes  de chamar a troca x = %15.5lf e y = %15.5lf", x, y);
 troca (&x, &y);
 printf("\n* depois de chamar a troca x = %15.5lf e y = %15.5lf", x, y);

 /* x e y antes e depois da troca usando ponteiros */
 *xx = 1.23456;
 *yy = 6.54321;
 printf("\n\n\n************** segundo exemplo  ******************");
 printf("\n* antes  de chamar a troca x = %15.5lf e y = %15.5lf", x, y);
 troca (xx, yy);
 printf("\n* depois de chamar a troca x = %15.5lf e y = %15.5lf", x, y);
}
```

A seguir o que será impresso por esse programa:

```
************** primeiro exemplo ******************
* antes  de chamar a troca x =         2.78000 e y =         3.14000
* depois de chamar a troca x =         3.14000 e y =         2.78000


************** segundo exemplo  ******************
* antes  de chamar a troca x =         1.23456 e y =         6.54321
* depois de chamar a troca x =         6.54321 e y =         1.23456
```

Veja outro exemplo abaixo que usa a função troca. Dados 3 números, imprima-os em ordem crescente.

```
/* Dados 3 números x, y e z, imprimí-los em ordem crescente */
int main() {
 double x, y, z;
 /* ler os 3 números */
 printf("\n\n\n entre com x, y e z separados por brancos:");
 scanf("%lf%lf%lf", &x, &y, &z);

 /* inverte a com b e b com c se for o caso */
 if (x > y) troca (&x, &y);
 if (y > z) troca (&y, &z);
 if (x > y) troca (&x, &y);

 /* imprime em ordem crescente */
 printf("\n\n\n************** ordem crescente ******************");
 printf("\n %lf   %lf   %lf", x, y, z);
}
```

Veja o que será impresso:

```
 entre com x, y e z separados por brancos:3.21 4.567 3.5678


************** ordem crescente ******************
 3.210000   3.567800   4.567000
```

Chamadas de funções dentro de funções

Pode haver chamadas de funções dentro de funções. Ou seja, uma função pode chamar outras funções. Veja o exemplo abaixo:

```
int f(double x)
:
:
double g(int a, int b) {
 double s1, s2;
 :
 :
 s1 = f(s1) + f(s2);
 :
 :
}
int h(int k, double z) {
 double xx; int i;
 :
 :
 xx = g(5)*g(0) + g(i);
 i = f (xx);
}
int main() {
 int zz;
 :
 :
 zz = h(5, 2.34);
 :
 :
}
```

Note que o programa principal chama h. A função h chama g e f. A função g chama f.

No exemplo abaixo a função **`imprimemaior (a, b, c)`** , imprime o maior entre **`a`**, **`b`** e **`c`**, usando a função **`troca`**.

```
void imprimemaior(double a, double b, double c)
   /* imprime o maior entre a, b e c */
   if (a > b) troca (&a, &b);
   if (b > c) troca (&b, &c);
   if (a > b) troca (&a, &b);
   printf("\n**** ordem crescente:%lf  %lf  %lf", a, b, c);
}
```

Exercício: Usando a função media_final , escreva uma função estado_final, que receba a média final e devolva:
-1 se o aluno foi reprovado, isto é, média final < 3
0 se o aluno está de recuperação, isto é, 3 <= média final < 5
1 se o aluno está aprovado, isto é, média final >= 5

Declaração de protótipos de funções

Considere as funções f, g e h abaixo com os seguintes protótipos:

```
int f(char a[], int n)
void g(int a[], int n, float x)
double h(char a[], char b[], int n)
```

Supondo agora que f chama g, g chama h e h chama f. Qual a ordem da declaração? Qualquer que seja a ordem escolhida será necessário referenciar uma função sem que ela tenha sido declarada.

A solução é fazer a declaração do protótipo das funções e declarar a função abaixo da seguinte forma:

```
int f(char a[], int n) {
  void g(int *, int, float);
  ......
}

void g(int a[], int n, float x) {
   void h(char *, char *, int);
   ......
}

double h(char a[], char b[], int n) {
   void f(char *, int);
   .......
}
```

Outra utilidade é permitir que o programa principal use as funções antes de sua declaração:

```
int main() {
 void f(char *, int);
 void g(int *, int, float);
 void h(char *, char *, int);
 .........
}

int f(char a[], int n)
  ....
void g(int a[], int n, float x)
  ....
double h(char a[], char b[], int n)
  ....
```

Funções e Macros

Já vimos que o pré-processador da linguagem c, possui alguns comandos como o #define para definir sinônimos de nomes e o #include para incluir arquivos num programa fonte.

Em particular, com o #define é possível a definição de estruturas chamadas de macro instruções, um pouco mais complexas que um simples sinônimo de um nome. Vejamos exemplos:

```
#define max(a, b) (a > b) ? a : b
```

Se no programa ocorrer um comando do tipo z = max(x, y); o mesmo será substituido por:

```
z = (x > y) ? x : y;
```

Para evitar problemas com prioridade de operadores melhor seria a definição:

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

Dessa forma um comando do tipo z = max(x*y/(w+1), x/(x-w)) seria substituido por:

```
z = ((x*y/(w+1)) > (x/(x-w)) ? (x*y/(w+1)) : (x/(x-w)))
```

De fato, os parêntesis merecem um cuidado especial. Suponha a seguinte definição:

```
#define quad(x) x*x
```

que calcula o quadrado de x. Se ocorrer um comando do tipo z = quad(a+b) o mesmo não calculará o quadrado de a+b, pois será susbtituido por:

```
z = a+b*a+b;
```

A definição correta seria:

```
#define quad(x) (x)*(x)
```

O mais importante é não confundir essas macros, que são simplesmente substituição de texto com a definição e uso de uma função. Observe que quando uma função é chamada, ocorre um desvio para o ponto onde está a função, e ao final da sua execução o retorno ao ponto em que foi chamada.

Exercício: Defina uma macro troca(a, b) que troca o valor das duas variáveis a e b. Compare com uma função troca que faz a mesma coisa.

Funções como parâmetros de outras funções

Uma função pode ser passada como parâmetro de outra função.
Suponha por exemplo uma função GRAF que traça o gráfico de uma função no intervalo [a,b] com n pontos. Para que GRAF possa traçar o gráfico de qualquer função, a função também deveria ser um parâmetro.

Declaração e uso de funções como parâmetro.

A função `ImpFunc` abaixo, imprime os valores de uma função `double f`, com parâmetro `double`, no intervalo `[-1,1]`, com passo `0.1`.
Veja a sintaxe da declaração e da chamada da função.

```
void ImpFunc(double (*f) (double)) {
     double t,y=-1.0;
     while (y<=1.0) {
       t=(*f)(y);
       printf("\n%lf - %lf",y,t);
       y=y+0.1;
     }
}
```

Observe que são especificados apenas os tipos dos parâmetros da função parâmetro.
Assim, se `f` tivesse dois parâmetros `double` e um `int` a declaração ficaria:

```
void ImpFunc(double (*f) (double,double,int)) {
```

Observe também a chamada da função `(*f)(y)` . Os parêntesis são necessários devido à prioridade dos operadores.

Como exemplo, o programa abaixo chama a função `ImpFunc` com várias funções diferentes como parâmetros.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double ff(double a) {
    return a*a;
}

double gg(double a) {
    double b;
    if (a==0) b=0;
    else b=fabs(a)/a;
    return b;
}

void ImpFunc(double (*f) (double)) {
    double t,y=-1.0;
    while (y<=1.0) {
      t=(*f)(y);
      printf("\n%lf - %lf",y,t);
      y=y+0.1;
    }
}

int main() {
  printf("\nFuncao ff");
  ImpFunc(ff);
  printf("\nFuncao gg");
  ImpFunc(gg);
  printf("\nFuncao fabs");
  ImpFunc(fabs);
  printf("\nFuncao seno");
  ImpFunc(sin);
  printf("\nFuncao cosseno");
  ImpFunc(cos);
  system("PAUSE");
```

```
  return 0;
}
```

Funções com número variável de parâmetros

Já vimos que as funções **printf()** e **scanf()** são funções do C, definidas na biblioteca **<stdio.h>**.
Essas funções tem uma particularidade. Possuem um número variável de parâmetros.
Como então devemos fazer para definir uma função com um número variável de parâmetros?
Exemplo: vamos construir uma função que imprime uma quantidade qualquer de valores inteiros.

Em primeiro lugar, o primeiro parâmetro deve de alguma forma definir a quantidade de parâmetros da função. As seguintes diretrizes devem ser seguidas:

1) Sempre incluir a biblioteca **stdarg.h** do C para usar as funções de parâmetros variáveis.

2) Existem 3 funções e um tipo de dado que irão ser necessárias dentro da função na qual vai querer ter parâmetros indefinidos, São elas:

```
void va_arg(va_list argptr,tipo de dado do parâmetro variável);
type va_start(va_list argptr,nome do último parâmetro conhecido);
void va_end(va_list argptr);
```

3) Declarar a função com pelo menos um parâmetro conhecido (para o controle do número de argumentos variáveis que irão ser passados na função);

4) Sempre encerrar a função com a função va_end();

Exemplo de programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h> /* Sempre inclua essa biblioteca */

/* Imprime Valores Inteiros
   O primeiro parâmetro define de alguma maneira a quantidade de parâmetros */
void ImpValInteiros(int NumeroDeParametros,...) {
  int i,num;
  /* Variável tipo va_list - variável do tipo lista de parâmetros. */
  va_list ListaDeParametros;
  /* A função va_start inicia o processamento da lista pelo primeiro elemento.
     Neste caso, atribui o valor do primeiro parametro à uma variável int */
  va_start(ListaDeParametros,NumeroDeParametros);
  /* Imprime todos os parâmetros */
  for(i=1; i<=NumeroDeParametros;i++) {
     /* A cada chamada va_arg retorna valor do próximo parâmetro baseado no tipo */
     num=va_arg(ListaDeParametros,int);
     printf("%d\n",num);

  }
  /* Sempre terminar com va_end no final. */
```

```
  va_end(ListaDeParametros);
  return;
}

int main() {
  int a=10,b=2;
  ImpValInteiros(5,1,3,5,7,9);
  system("PAUSE");
  ImpValInteiros(4,0,2,4,6);
  system("PAUSE");
  ImpValInteiros(4,a,b,a+b,a-b);
  system("PAUSE");
  ImpValInteiros(6,a,b,a+b,a-b,a*b,a/b);
  system("PAUSE");
  return 0;
}
```

O programa acima imprime:

```
1
3
5
7
9
Pressione qualquer tecla para continuar. . .
0
2
4
6
Pressione qualquer tecla para continuar. . .
10
2
12
8
Pressione qualquer tecla para continuar. . .
10
2
12
8
20
5
Pressione qualquer tecla para continuar. . .
```

Diga agora com as funções printf() e scanf() descobrem a quantidade de parâmetros que serão lido ou impressos?