



Introdução à Programação

Capítulo V

Funções

Engenharia Informática e de Sistemas

Introdução

As funções:

- ajudam-nos a dividir tarefas complicadas noutras mais simples;
- facilitam a construção de novos programas utilizando funções já escritas por alguém;
- podem também esconder detalhes (que por vezes não nos interessa conhecer) clarificando o programa e facilitando futuras modificações.

Um programa em C é normalmente constituído por inúmeras pequenas funções (em vez de poucas e grandes).

Não pode haver funções dentro de outras funções no C.

As funções podem estar em qualquer ordem num ficheiro ou encontrarem-se divididas por vários ficheiros.

Embora ainda não se tenha abordado a escrita de funções em **C**, já se têm utilizado algumas nos programas apresentados nos capítulos anteriores. São exemplos as funções ***printf***, ***scanf***, ***getchar***,... , que fazem parte da biblioteca *standard* do **C**.

Exercício 1:

Desenvolva um programa que produza o seguinte *output* no ecrã:

```
*****
Números entre 1 e 5
*****
1
2
3
4
5
*****
```

Resolução:

```
#include <stdio.h>

void main()
{
    int i;
    for (i=1;i<=20;i++)
        putchar( '*' );
    putchar( '\\n' );
    printf("Números entre 1 e 5\\n");
    for (i=1;i<=20;i++)
        putchar( '*' );
    putchar( '\\n' );
    for (i=1;i<=5;i++)
        printf("%d\\n",i);
    for (i=1;i<=20;i++)
        putchar( '*' );
    putchar( '\\n' );
}
```

O código utilizado para escrever uma linha de asteriscos no ecrã

```
for (i=1;i<=20;i++)  
    putchar( '*' );  
putchar( '\\n' );
```

encontra-se **repetido três vezes**.

O ideal seria escrever este pedaço de código **apenas uma única vez** e invocá-lo sempre que necessário.

A solução consiste em **dividir o programa em pequenos fragmentos de código**, cada um dos quais responsável por uma determinada tarefa.

Exercício 2:

Desenvolva um programa que escreva uma linha de 20 asteriscos no ecrã.

Resolução:

```
#include <stdio.h>  
void main()  
{  
    int i;  
    for (i=1;i<=20;i++)  
        putchar( '*' );  
    putchar( '\\n' );  
}
```

O programa anterior escreve no ecrã uma linha com 20 asteriscos e foi implementado como sendo a função **main**.

Como a sua tarefa é escrever uma linha, em vez de lhe darmos o nome de **main**, vamos dar-lhe o nome de **linha**.

```
#include <stdio.h>
void linha()
{
    int i;
    for (i=1; i<=20; i++)
        putchar( '*' );
    putchar( '\\n' );
}
```

É claro que se tentarmos executar este código vamos obter um erro de *linkagem*, uma vez que a função **main** não se encontra presente (e é sempre a partir desta que se inicia a execução de qualquer programa em C).

O que se fez anteriormente foi escrever o código responsável pela escrita de uma linha de asteriscos no ecrã. O facto de a função **linha** existir, não significa que venha a ser executada.

É a função ***main()*** (ou qualquer outra função invocada pela função ***main()***) que terá que solicitar os serviços desta função. Para tal basta escrever o nome da função com os respectivos parêntesis (***linha();***).

Usando a função ***linha()***, a resolução do **Exercício 1** pode ser escrita como:

```
#include <stdio.h>

void linha()
{
    int i;
    for(i=1;i<=20;i++)
        putchar( '*' );
    putchar( '\\n' );
}

void main()
{
    int i;
    linha();          /* Escreve uma linha de 20 asteriscos */
    printf("Números entre 1 e 5\\n");
    linha();          /* Escreve uma linha de 20 asteriscos */
    for(i=1;i<=5;i++)
        printf("%d\\n",i);
    linha();          /* Escreve uma linha de 20 asteriscos */
}
```

O programa apresentado atrás contém duas funções escritas no mesmo ficheiro:

- A função **main()** é responsável por iniciar o programa e executar todas as instruções presentes no seu interior.
- A função **linha()** é responsável por escrever uma linha de asteriscos no ecrã. Sempre que se pretender escrever uma tal linha no ecrã bastará invocar a referida função, evitando deste modo a repetição da escrita do código que a mesma executa.

Concluindo:

Uma função é simplesmente um conjunto de instruções agrupadas sob um determinado nome.

As funções são os “blocos de construção” de um programa em linguagem C.

Cada função é essencialmente um pequeno programa, com as suas próprias declarações e instruções.

Usando funções pode-se dividir um programa em pequenas partes mais fáceis de compreender e de modificar.

As funções evitam duplicações de código e são reutilizáveis: pode pegar-se numa função que originalmente fazia parte de um programa e usá-la noutros programas.

Um programa em C possui obrigatoriamente uma função *main()*, independentemente do número e da variedade de outras funções que contenha.

Características de uma função

- Cada função tem um nome único que serve para a sua invocação;
- Pode ser invocada a partir de outras funções;
- Deve realizar uma tarefa bem definida;
- Deve comportar-se como uma caixa negra – não interessa como funciona mas que o resultado final seja o esperado;
- O código de uma função deve ser o mais independente possível do resto do programa e também o mais genérico possível, de forma a poder ser reutilizado noutros programas;
- Pode receber parâmetros que alterem o seu comportamento de forma a adaptar-se facilmente a situações distintas;
- Pode retornar um valor como resultado da sua execução.

Definição de uma função

```
tipo_de_retorno nome_da_função( parâmetros )           /* cabeçalho */  
{  
    Declarações / Definições                               /* corpo da função */  
    instruções  
}
```


Modo de funcionamento

- O código de uma função só é executado quando esta é invocada algures no programa ao qual está ligada;
- Quando uma função é invocada, o programa que a invoca é temporariamente “suspenso” enquanto são executadas as instruções que fazem parte do corpo da função. Uma vez terminada a função, o controlo de execução do programa regressa ao local em que esta foi invocada;
- O programa que invoca a função pode enviar **argumentos** que são recebidos e armazenados pela função, em variáveis locais automaticamente inicializadas com os valores enviados. A estas variáveis locais dá-se o nome de **parâmetros** da função;
- Depois de executada, uma função pode devolver um valor para o programa que a invocou.

Exercício 3:

Desenvolva um programa que, recorrendo a três funções distintas, produza o seguinte *output* no ecrã:

```
***
*****
*****
*****
***
```

Resolução:

```
#include <stdio.h>
void linha3()
{
    int i;
    for(i=1;i<=3;i++)
        putchar('*');
    putchar('\n');
}
void linha5()
{
    int i;
    for(i=1;i<=5;i++)
        putchar('*');
    putchar('\n');
}
```

```
void linha7()
{
    int i;
    for(i=1;i<=7;i++)
        putchar('*');
    putchar('\n');
}
void main()
{
    linha3();
    linha5();
    linha7();
    linha5();
    linha3();
}
```

Na resolução do exercício anterior escrevemos quatro funções:

- **linha3**: função responsável por escrever 3 asteriscos no ecrã;
- **linha5**: função responsável por escrever 5 asteriscos no ecrã;
- **linha7**: função responsável por escrever 7 asteriscos no ecrã;
- **main**: função que chama as outras funções.

Observando o código das três funções, verifica-se que é todo igual exceto nas seguintes linhas:

```
for (i=1; i<=3; i++)
```

```
for (i=1; i<=5; i++)
```

```
for (i=1; i<=7; i++)
```

O que varia é apenas o nº de asteriscos a apresentar no ecrã.

O ideal é desenvolver uma única função que escreva uma linha no ecrã com um determinado número de asteriscos, número esse a especificar em cada chamada à função.

Assim, se quisermos uma linha com **3** asteriscos chamamos **linha(3)**, se quisermos uma linha com **5** asteriscos chamamos **linha(5)**, se quisermos uma linha com **100** asteriscos chamamos **linha(100)**, etc.

Usando a nova função ***linha()***, a função ***main()*** do exemplo anterior transforma-se em:

```
void main()  
{  
    linha(3);  
    linha(5);  
    linha(7);  
    linha(5);  
    linha(3);  
}
```

Por sua vez, a função ***linha()*** terá as seguintes características:

Nome: ***linha()***

Nº de parâmetros: 1

Tipo do parâmetro: ***inteiro***

Nome do parâmetro: ***num***

O cabeçalho será então: ***void linha(int num)***

O corpo da função será igual ao das anteriores exceto na condição do ciclo:

```
for (i=1; i<=num; i++)
```

Deste modo, a resolução do Exercício 3 pode ser novamente escrita como:

```
#include <stdio.h>

void linha(int num)
{
    int i;
    for(i=1;i<=num;i++)
        putchar('*');
    putchar('\n');
}

void main()
{
    linha(3);
    linha(5);
    linha(7);
    linha(5);
    linha(3);
}
```

Parâmetros / Argumentos

A comunicação com uma função faz-se através dos **argumentos** que lhe são enviados e dos **parâmetros** (presentes na função) que os recebem.

O número de **parâmetros** de uma função depende das necessidades do programador (pode ser 0,1,2,...).

Os **parâmetros** são separados por vírgula e é absolutamente necessário que, para cada um deles, seja indicado o seu tipo:

```
funcao (int x, char y, float k, double w)
```

Um **parâmetro** não é mais do que uma variável local à função a que pertence, a qual é automaticamente inicializada com o valor enviado pelo programa que chama a função.

Em C todos os **argumentos** de funções **são passados por valor**. À função que é chamada, é dada uma cópia dos valores dos **argumentos** e ela cria outras variáveis temporárias (os **parâmetros**) para armazenar estes valores. ➔ Uma função chamada não pode alterar **diretamente** o valor de uma variável da função que a chama, pode apenas alterar a sua cópia temporária.

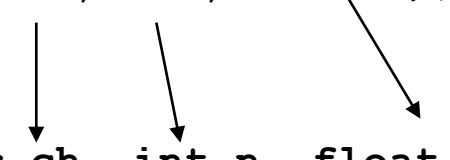
A passagem de **argumentos** para uma função, deve ser realizada colocando-os dentro de parêntesis, separados por vírgulas, imediatamente a seguir ao nome da função.

Na chamada a uma função, o número e o tipo dos **argumentos** enviados devem ser coincidentes com os dos **parâmetros** presentes no cabeçalho da função.

Exemplo:

```
main()
{
    ...
    funcao('A', 123, 23.456);
    ...
}

funcao(char ch, int n, float x)
{
    ...
}
```



Se a função tiver mais do que um parâmetro, os **argumentos** enviados são associados aos **parâmetros** pela ordem em que são escritos.

É muito comum chamar-se **parâmetros** quer aos **argumentos** de invocação quer aos verdadeiros **parâmetros** da função.

Nome de uma função

A escolha do nome de uma função obedece às regras anteriormente apresentadas para os nomes das variáveis.

O nome deve ser único (não pode ser igual ao de outra função ou ao de uma variável) e para além disso deve ser sugestivo relativamente à tarefa desempenhada pela função.

Tipo de retorno

É o tipo do valor que uma função devolve, o qual obedece às seguintes regras:

- As funções só não podem devolver *arrays*, não existindo quaisquer outras restrições quanto ao tipo devolvido;
- Se o *tipo_de_retorno* for omitido na definição da função, é assumido que esta devolve um valor inteiro (*int*);
- Se o *tipo_de_retorno* for *void*, significa que a função não devolve nenhum valor.

A instrução **return**

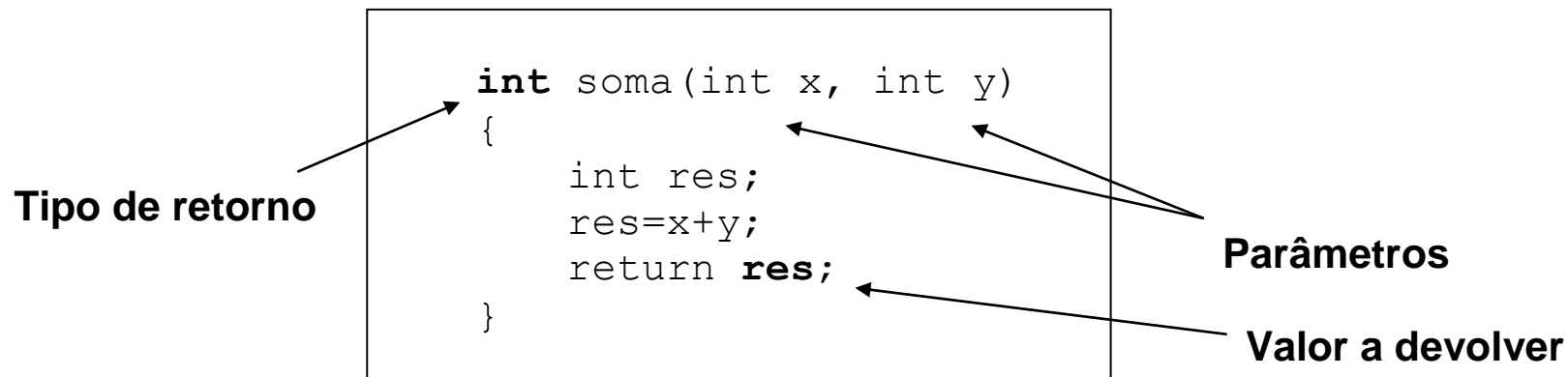
A instrução **return** permite terminar a execução de uma função e voltar ao programa que a invocou. (Se esta instrução for executada na função *main* faz com que o programa termine.)

A devolução de um resultado por uma função, é feita através da instrução **return** seguida do valor a devolver.

Suponhamos que pretendíamos calcular a soma de dois números inteiros. A função responsável por tal tarefa deve ter a capacidade de realizar a soma e de devolver o resultado desta.

```
n=soma(3,4);  
printf("%d\n",n);    /*7 */
```

A função *soma* terá que receber dois valores inteiros e terá que devolver um resultado do mesmo tipo, que corresponderá à soma dos dois parâmetros.



Declaração de uma função

Uma função deve ser declarada antes de ser invocada.

A declaração proporciona ao compilador um breve conhecimento sobre a função cuja definição aparecerá mais tarde. Esta é feita da seguinte forma:

```
tipo_de_retorno nome_da_função( parâmetros );
```

A declaração anterior é conhecida por **protótipo da função** e corresponde ao cabeçalho desta, seguido por um ponto e vírgula (;).

Um **protótipo** faculta uma descrição completa sobre como chamar a função: quantos argumentos enviar, quais deverão ser os seus tipos e qual o tipo de resultado que será devolvido.

Passagem de parâmetros a funções

Em C todos os **argumentos** de funções **são passados por valor**.

À função que é chamada, é dada uma cópia dos valores dos **argumentos** e ela cria outras variáveis temporárias (os **parâmetros**) para armazenar estes valores. ➔ Uma função chamada não pode alterar **diretamente** o valor de uma variável da função que a chama, pode apenas alterar a sua cópia temporária.

Ou seja, uma variável passada por valor a uma função não é modificada pela execução da função.

```

#include <stdio.h>

void incrementa(int num)
{
    printf("Em incre: antes %3d\n", num);
    num++;
    printf("Em incre: depois %3d\n", num);
}

void main(void)
{
    int numero;

    printf("Diga um valor inteiro:");
    scanf("%d", &numero);
    printf("\nEm main: antes %3d\n", numero);
    incrementa(numero);
    printf("\nEm main: depois %3d\n", numero);
}

```

Possível resultado de execução:

Diga um valor inteiro:6

Em main: antes 6

Em incre: antes 6

Em incre: depois 7

Em main: depois 6

Ponteiros e endereços

Considere-se uma variável inteira i ($\text{int } i$). A expressão:

$\&i$

fornece a **referência (ou o endereço) dessa variável**; isto é algo que permite identificar e aceder a essa variável. Fisicamente a referência é o número (endereço) da posição de memória que a variável ocupa (e através do qual se lhe pode aceder).

Um **ponteiro** (ou apontador) é uma **variável que contém o endereço de uma outra variável**.

Se pi for um ponteiro para uma variável inteira ($\text{int } *pi$) então podemos fazer a atribuição:

$pi = \&i$

significando que colocamos em pi um valor que é a referência ou endereço da variável i .

A manipulação de variáveis através de ponteiros é possibilitada pelos operadores "&" e "*".

- O operador & aplica-se a uma variável e fornece o endereço dessa variável
- O operador * aplica-se a um endereço e fornece o valor da variável que esse endereço refere.

Exemplo:

```
main()
{
    ...
    int x = 9;      /* a variável inteira x fica com valor 9 */
    int * px;       /* px é um ponteiro para inteiro*/

    px = &x; /* atribui-se a px o valor do endereço da variável x */
    *px = 12 /* x fica com valor 12 */

    ...
}
```

Passagem de parâmetros por referência

Os **ponteiros** têm muitas utilizações na linguagem C. Uma delas é **permitirem implementar a passagem de parâmetros** a funções **por referência**, isto é, de forma que a função receba e possa modificar as próprias variáveis passadas como argumento.

Quando um **parâmetro** é **passado** a uma função **por referência**, o parâmetro formal correspondente **recebe a localização na memória da variável atual** em vez do seu valor.

Assim todas as instruções que forem executadas sobre o parâmetro formal dentro da função são, de facto, executadas sobre a variável atual.

```

#include <stdio.h>

#include <stdio.h>

void incrementa(int *num)
{
    printf("Em incre: antes %3d\n", *num);
    (*num)++;
    printf("Em incre: depois %3d\n", *num);
}

void main(void)
{
    int numero;

    printf("Diga um valor inteiro:");
    scanf("%d", &numero);
    printf("\nEm main: antes %3d\n", numero);
    incrementa(&numero);
    printf("\nEm main: depois %3d\n", numero);
}

```

Possível resultado de execução:

Diga um valor inteiro:21

Em main: antes 21

Em incre: antes 21

Em incre: depois 22

Em main: depois 22

Variáveis locais (ou automáticas)

Quando uma variável é definida no corpo de uma função, diz-se que é **local** a essa função.

Exemplo:

```
int  soma(int x, int y)
{
    int res;          /* variável local */

    res=x+y;
    return res;
}
```

Por defeito, as **variáveis locais** têm as seguintes propriedades:

- Quando a função a que pertencem é chamada, é “automaticamente” alocado espaço em memória para as armazenar. Assim que a execução da função termina, este espaço é libertado. Por esta razão estas variáveis são também chamadas de **variáveis automáticas**.
- Estas variáveis são apenas **visíveis dentro da função** onde são definidas (desde o local da definição até ao final do corpo da função). Por esta razão é possível que outras funções usem variáveis com os mesmos nomes para outros fins.

Variáveis estáticas

Colocando a palavra **static** na definição de uma variável, a variável passa a ter um armazenamento permanente (passa a ser estática).

As variáveis estáticas podem ser globais (definidas fora das funções) ou locais (definidas dentro das funções).

As funções podem também comunicar através de **variáveis estáticas** (globais ou locais)

Exemplo:

```
static int j=2;

void f1()
{
    static int i=6;
    ...
}
...
void fk()
{
    ...
}
```

Como as variáveis **i** e **j** (no exemplo anterior) foram definidas como estáticas, irão ocupar a mesma localização de memória durante toda a execução do programa.

Quando a função **f1()** terminar, a variável **i** não vai perder o seu valor. Esta variável **i** pode ser utilizada para transmitir informação entre diferentes execuções da função **f1()** durante toda a execução do programa (sendo apenas inicializada da 1ª vez que a função **f1()** for executada).

*Uma variável **estática local** possui **memória privada e permanente numa função**.*

A variável **j** é visível em todas as funções definidas abaixo da sua definição no ficheiro do programa (por exemplo na função **f1()**, ... na função **fk()**) mas não em nenhum outro ficheiro, não perdendo o seu valor durante toda a execução do programa. Esta variável **j** pode ser utilizada para transmitir informação entre todas estas funções que estão no ficheiro.

Uma variável **estática global** é útil para **esconder nomes** que devem ser **externos** para serem partilhados, mas que não devem ser visíveis a outros ficheiros para não haver hipótese de conflitos (podem existir variáveis com o mesmo nome noutros ficheiros).

*Uma variável **estática global** possui **memória privada e permanente para as funções de um ficheiro**.*

Parâmetros

Têm as mesmas propriedades que as variáveis locais.

A única diferença é que os parâmetros são automaticamente inicializados (com os valores dos argumentos ou parâmetros reais) de cada vez que a função é chamada.

Variáveis globais (ou externas)

As funções podem igualmente comunicar através de **variáveis globais** ou **externas**: são variáveis que são definidas fora do corpo de qualquer função.

As **variáveis externas** têm as seguintes propriedades:

- Têm armazenamento permanente;
- São visíveis em todo o ficheiro (desde o local da sua definição até ao fim do mesmo). Logo, qualquer variável global pode ser acedida por todas as funções que se encontrem a seguir à sua definição.
- Podem ser declaradas, se necessário, tornando-se visíveis noutros ficheiros (desde o local da sua declaração até ao fim do mesmo), ou no próprio ficheiros (se a definição surgir depois da declaração). Assim sendo, qualquer variável global pode ser acedida por todas as funções dum programa, mesmo que estejam em diferentes ficheiros, desde que exista uma declaração prévia nesse ficheiro (ou seja, se encontrem a seguir à sua declaração).

REGRAS DE VALIDADE:

As **funções** e **variáveis** globais que compõem um programa em C não precisam de ser compiladas todas ao mesmo tempo: o texto do programa pode ser distribuído por vários ficheiros e rotinas previamente compiladas podem ser carregadas das bibliotecas.

A **validade de um nome** (de uma variável, função, ...) tem a ver com a **parte do programa no qual esse nome pode ser utilizado**.

Uma **variável automática** definida no início de uma função **é válida apenas nessa função** e não está relacionada com variáveis do mesmo nome que possam existir noutras funções. O mesmo acontece com os **parâmetros das funções**.

A validade de uma **variável global** (e de uma **função**) tem início no **ponto em que ela é definida** num ficheiro e vai **até ao fim desse ficheiro**.

Se uma **variável externa** for **referida antes de ter sido definida**, ou se **está definida noutro ficheiro**, então uma **declaração extern** é obrigatória.

É preciso distinguir:

- * **declaração** de uma variável externa
-> **anuncia as propriedades de uma variável** (tipo, tamanho, etc);
- * **definição**
-> que além disso, **aloca memória** para a variável (e inicializa-a).

Se as **definições**:

```
float x, y;  
double aa;
```

aparecem fora de qualquer função, definem as variáveis externas x, y como do tipo float e a variável **aa** como do tipo double (definem o tipo e consequentemente o tamanho, e provocam a respetiva alocação de espaço para elas, inicializando-as a zero) **e servem de definição para o resto do ficheiro.**

As linhas

```
extern float x, y;  
extern double aa;
```

declaram para o resto do ficheiro x, y como do tipo float e **aa** como do tipo double, mas não criam as variáveis nem lhes reservam memória.

Uma **variável externa só pode ser definida uma vez** num programa, embora possa haver vários ficheiros com declarações **extern** para aceder à variável (pode existir uma declaração mesmo no ficheiro que contém a definição!). Qualquer inicialização só pode ser feita na definição.

Se num ficheiro, tivermos uma função com uma **variável automática ou um parâmetro** com o **mesmo nome** de uma **variável externa**, então **dentro da função** o que é reconhecido é o parâmetro ou a variável interna, enquanto que fora da função esse nome refere-se à variável externa.

Exemplos

Exemplo 1

```
/* Função que calcula e devolve a média de dois números reais passados  
como argumento */
```

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

```
/* Definição da função média */  
/* Tem dois parâmetros reais */  
/* Devolve um resultado real */
```

```
float media(float a, float b)  
{  
    float valor;  
  
    printf("Funcao media em accao\n");  
    printf("Parametros: %f\t%f\n", a, b);  
    valor = (a + b)/2;  
    return valor;  
}
```

```
/* Exemplo de utilização da função media */  
int main()  
{  
    float x = 4.5, y = 3.8, z;  
  
    printf("Antes de chamar a funcao!\n");  
    z = media(x, y);  
    printf("A funcao foi chamada e produziu o resultado %4.2f\n", z);  
    return 0;  
}
```

Resultado da execução:

```
Antes de chamar a funcao!  
Funcao media em accao  
Parametros: 4.500000    3.800000  
A funcao foi chamada e produziu o resultado 4.15
```

Exemplo 2

```
/* Função para calcular e devolver a média de dois números reais e
função para devolver o menor de dois números passados como argumento */

#include <stdio.h>

/* Função que devolve o menor de dois números passados como argumento */
float menor(float m, float n)
{
    if(m < n)
        return m;
    else
        return n;
}

/* Função que devolve a média de dois números reais */
float media(float a, float b)
{
    float valor;
    valor = (a + b)/2;
    return valor;
}
```

```
/* Exemplo de utilização das funções definidas anteriormente */
```

```
int main()  
{  
    float x = 4.5, y = -2.5, z, w;  
    z = menor(x, 5);  
    w = media(z, y);  
    printf("%f\t%f\t%f\n", z, w, media(x, y));  
    return 0;  
}
```

Resultado da execução:

4.500000	1.000000	1.000000
----------	----------	----------

Exemplo 3

```
/* Programa para escrever um quadrado de asteriscos */

#include <stdio.h>

/* Função que escreve uma linha de asteriscos. A dimensão da linha é
passada como argumento. A função não devolve nenhum resultado
explicitamente. */

void linha_asteriscos(int tamanho)
{
    int i;
    for(i=0; i<tamanho; i++)
        putchar('*');
}
```

```
/* Função que obtém um valor inteiro entre 1 e 10. A função não recebe  
nenhuma informação como argumento. */
```

```
int obtem_lado(void)  
{  
    int i;  
    do{  
        printf("Lado: ");  
        scanf("%d", &i);  
    }while(i < 1 || i > 10);  
    return i;  
}
```

```
/* Função principal do programa. Escreve um quadrado de asteriscos. O
tamanho do lado é indicado pelo utilizador. */
```

```
int main()
{
    int i, lado;
    lado = obtem_lado();
    for(i=0; i<lado; i++)
    {
        linha_asteriscos(lado);
        putchar('\n');
    }
    return 0;
}
```

Exemplo de execução:

Lado: 0

Lado: 3

Exemplo 4

```
/* Função que calcula  $x^y$ . A base é um número real e o expoente é um valor inteiro. A função recebe dois argumentos (a base e o expoente) e devolve o resultado da operação  $x^y$ . */
```

```
/* Declaração vs. Definição de funções em linguagem C:  
É conveniente declarar uma função sempre que ela for utilizada  
(i.e., chamada) antes de ser definida (i.e., escrita) */
```

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

```
/* Declaração da função potencia. */
```

```
float potencia(float base, int exp);
```

```
/* Definição da função principal do programa. Nele pode ver-se um exemplo de utilização da função potencia */
```

```
int main()  
{  
    float x;  
    int y;  
    float res;
```



```

    printf("Indique a base e o expoente: ");
    scanf("%f%d", &x, &y);
    res = potencia(x, y);
    printf("O resultado e: %f\n", res);
    return 0;
}

/* Definição da função potencia */
float potencia(float base, int exp)
{
    int i;
    float total=1.0;

    for(i=1; i<=exp; i++)
        total = total*base;
    return total;
}

```

Exemplos de execução:

```

Indique a base e expoente: 4 5
O resultado e:1024

```

```

Indique a base e expoente: 2.3 4
O resultado e: 27.9841

```

Exemplo 5a

```
/* Passagem de argumentos por valor */

#include <stdio.h>

/* Função que troca os valores entre dois inteiros passados como
   argumento. */
void troca(int a, int b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
}

/* Exemplo de utilização da função troca. */
int main()
{
    int x = 10, y = 20;

    printf("Antes: X = %d\tY = %d\n", x, y);
    troca(x, y);
    printf("Depois: X = %d\tY = %d\n", x, y);
}
```

Resultado da execução:

Antes: X = 10 Y = 20
Depois: X = 10 Y = 20

Apesar do valor dos parâmetros ter sido trocado dentro da função, esse efeito não é visível fora. Isto sucede porque o que foi enviado na altura da chamada da função foi uma cópia do valor dos argumentos e não as próprias variáveis: em C os argumentos são passados por valor. As alterações que sofrem dentro das funções não são visíveis cá fora.

Isto permite-nos redefinir a função do exemplo 4 que calcula x^y .

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

    for( ; exp > 0; exp--)
        total = total*base;
    return total;
}
```

A alteração efetuada dentro da função ao parâmetro 'exp' não será visível fora, uma vez que estamos simplesmente a modificar a cópia local da função.

Exemplo 5b

```
/* Passagem de argumentos por referência */
#include <stdio.h>

/* Função que troca os valores entre dois inteiros passados como
argumento. */
void troca(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

/* Exemplo de utilização da função troca. */
int main()
{
    int x = 10, y = 20;
    printf("Antes: X = %d\tY = %d\n", x, y);
    troca(&x, &y);
    printf("Depois: X = %d\tY = %d\n", x, y);
}
```

Resultado da execução:

Antes: X = 10 Y = 20
Depois: X = 20 Y = 10

Exemplo 6

`/* Programa que controla a quantidade de líquido existente numa garrafa. Deve servir bebidas até a garrafa já não ter quantidade suficiente para mais nenhuma dose. No início, a garrafa tem 25 cl. */`

Versão 1: comunicação entre funções utilizando argumentos

```
#include <stdio.h>

#define DOSE_MAXIMA 5

int obtem_dose(void);
int serve(int quantidade, int dose);

int main()
{
    int garrafa = 25, dose;

    do{
        dose = obtem_dose();
        garrafa = serve(garrafa, dose);
        printf("Restam %d cl na garrafa\n", garrafa);
    }while(garrafa >= DOSE_MAXIMA);
    return 0;
}
```

```
int obtem_dose(void)
{
    int x;
    do{
        printf("Dose da bebida: ");
        scanf("%d", &x);
    }while(x < 1 || x > DOSE_MAXIMA);
    return x;
}
```

```
int serve(int qtd, int dose)
{
    printf("Servida mais uma bebida\n");
    return (qtd - dose);
}
```

Versão 2: comunicação entre funções utilizando uma variável global
--

```
#include <stdio.h>
#define DOSE_MAXIMA 5

int garrafa = 25;

int obtem_dose(void);
void serve(int dose);

int main()
{
    int dose;

    do{
        dose = obtem_dose();
        serve(dose);
        printf("Restam %d cl na garrafa\n", garrafa);
    }while(garrafa >= DOSE_MAXIMA);
    return 0;
}
```

```
int obtem_dose(void)
{
    int x;
    do{
        printf("Dose da bebida: ");
        scanf("%d", &x);
    }while(x < 1 || x > DOSE_MAXIMA);
    return x;
}
```

```
void serve(int dose)
{
    printf("Servida mais uma bebida\n");
    garrafa = garrafa - dose;
}
```


Versão 3: Comunicação de valores entre funções utilizando passagem de parâmetros por referência

```
#include <stdio.h>
#define DOSE_MAXIMA 5

int obtem_dose(void);
void serve(int *quantidade, int dose);

int main()
{
    int garrafa = 25, dose;

    do{
        dose = obtem_dose();
        serve(&garrafa, dose);
        printf("Restam %d cl na garrafa\n", garrafa);
    }while(garrafa >= DOSE_MAXIMA);
    return 0;
}
```

```
int obtem_dose(void)
{
    int x;

    do{
        printf("Dose da bebida: ");
        scanf("%d", &x);
    }while(x < 1 || x > DOSE_MAXIMA);
    return x;
}
```

```
void serve(int *qtd, int dose)
{
    printf("Servida mais uma bebida\n");
    *qtd = *qtd - dose;
}
```

Exemplo de execução (igual para as 3 versões):

Dose da bebida: 4

Servida mais uma bebida

Restam 21 cl na garrafa

Dose da bebida: 5

Servida mais uma bebida

Restam 16 cl na garrafa

Dose da bebida: 4

Servida mais uma bebida

Restam 12 cl na garrafa

Dose da bebida: 5

Servida mais uma bebida

Restam 7 cl na garrafa

Dose da bebida: 3

Servida mais uma bebida

Restam 4 cl na garrafa

Exemplo 7

```
/* Problemas que podem surgir quando se utilizam variáveis globais sem
ser necessário. A partilha da variável global i provoca um efeito
inesperado. */
```

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

```
#define LADO 5
```

```
int i;
```

```
void linha(void);
```

```
void quadrado(void);
```

```
int main()
```

```
{
```

```
    quadrado();
```

```
    return 0;
```

```
}
```

```
void linha(void)
```

```
{
```

```
    for(i = 0; i < LADO; i ++)
```

```
        putchar('*');
```

```
}
```

```
void quadrado(void)
{
    for(i=0; i<5; i++)
    {
        linha();
        putchar('\n');
    }
}
```

Exemplo de execução:

```
*****
```

O objetivo do programador era desenhar um quadrado de asteriscos no monitor. A única coisa que conseguiu foi desenhar uma linha!

Exemplo 8

```
/* Visibilidade de Variáveis */
```

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

```
int a = 10;
```

```
void f1(int i);
```

```
void f2(int i);
```

```
int main()
```

```
{
```

```
    f1(5);
```

```
    f2(1);
```

```
    a++;
```

```
    printf("%d\n", a);
```

```
    return 0;
```

```
}
```

```
void f1(int a)
```

```
{
```

```
    a++;
```

```
    printf("%d\n", a);
```

```
}
```

```
void f2(int x)
{
    int a = 5;

    a++;
    printf("%d\n", x);
}
```

Exemplo de execução:

```
6
1
11
```

Exemplo 9

`/* Visibilidade de variáveis */`

<pre>#include <stdio.h> int a = 4; void funcao(int b);</pre>	Visibilidade
<pre>int main() { int c = 1, d = 2; funcao(c); printf("%d\n", a++); funcao(d); return 0; }</pre>	a: global c, d: locais
<pre>void funcao(int e) { static int f = 5; int g = 2; printf("%d\t%d\t%d\t%d\n", a, e, f, g); f++; g++; }</pre>	a: global e: parâmetro f, g: locais

Resultado da execução:

4	1	5	2
4			
5	2	6	2

Exemplo 10

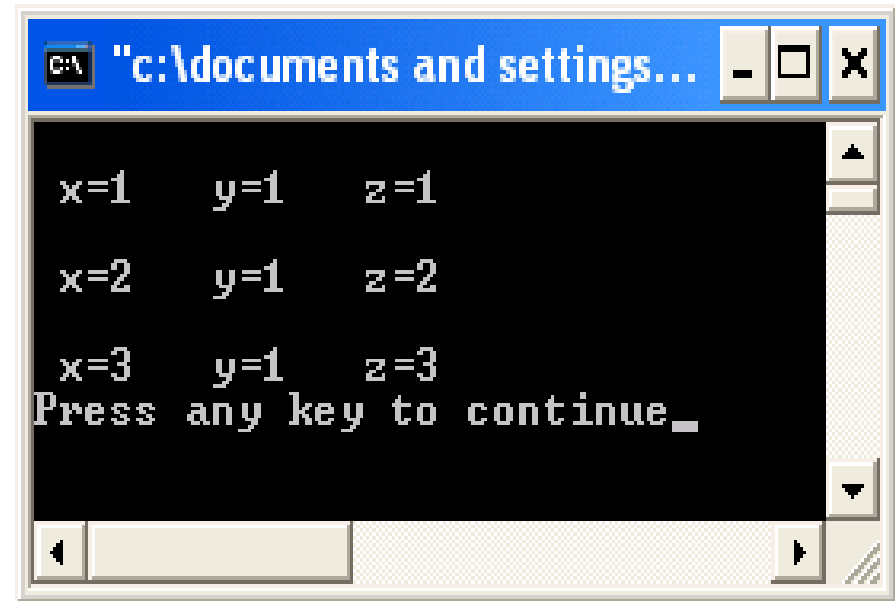
/* Qual o resultado da execução deste programa ? */

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

```
int x=1;  
void f(void);
```

```
void main()  
{  
    while (x<4)  
        f();  
}
```

```
void f(void){  
    int y=1;  
    static int z=1;  
    printf("\n x=%d    y=%d    z=%d \n", x++, y++, z++);  
}
```



Exemplo 11

`/* Qual o resultado da execução deste programa ? */`

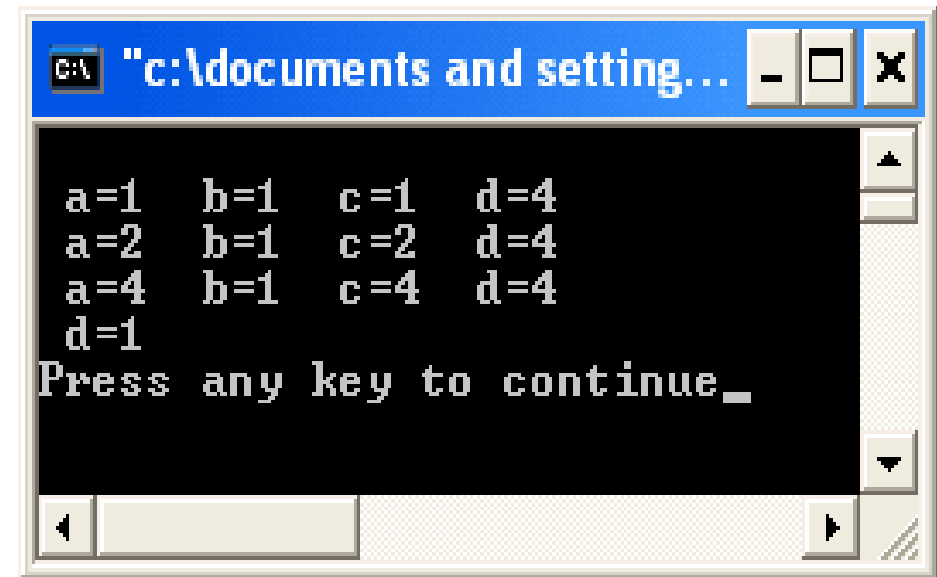
```
#include <stdio.h>
```

```
int a=1, d=1;
```

```
void duplica(void);
```

```
void main(void)    {  
    int i;  
    for ( i=0 ; i<3 ; i++)  
        duplica( );  
    printf("\n d=%d \n", d);  
  
}
```

```
void duplica(void){  
    int b=1, d=4;  
    static int c=1;  
    printf("\n a=%d  b=%d  c=%d  d=%d", a, b, c, d);  
    a*=2;  
    b*=2;  
    c*=2;  
  
}
```



Exemplo 12

`/* Qual o resultado da execução deste programa ? */`

`#include <stdio.h>`

`void f(int d);`

`int a=0;`

`void main()`

`{`

`int i;`

`for (i=0 ; i<3 ; i++)`

`f(0);`

`}`

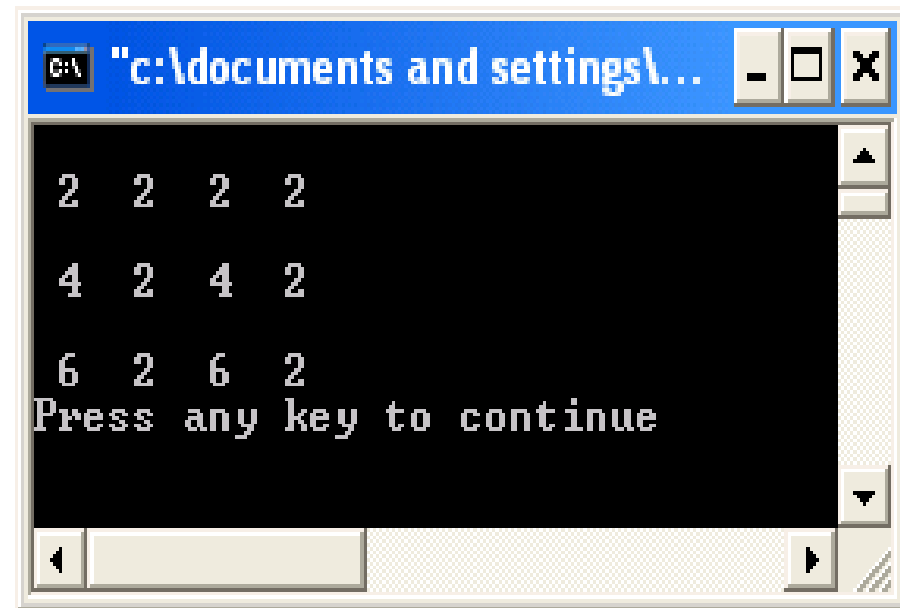
`void f(int d){`

`int b=0;`

`static int c=0;`

`printf("\n %d %d %d %d\n", a+=2, b+=2, c+=2, d+=2);`

`}`



Exemplo 13

`/* Qual o resultado da execução deste programa ? */`

`#include <stdio.h>`

`int ftotal(void);`

`int fsoma(void);`

`void main(void)`

`{`

`int i;`

`for (i=1 ; i<5 ; i++) printf("\n %d %d ",ftotal(), fsoma());`

`}`

`int ftotal(void)`

`{`

`static int total=0;`

`return total+=1;`

`}`

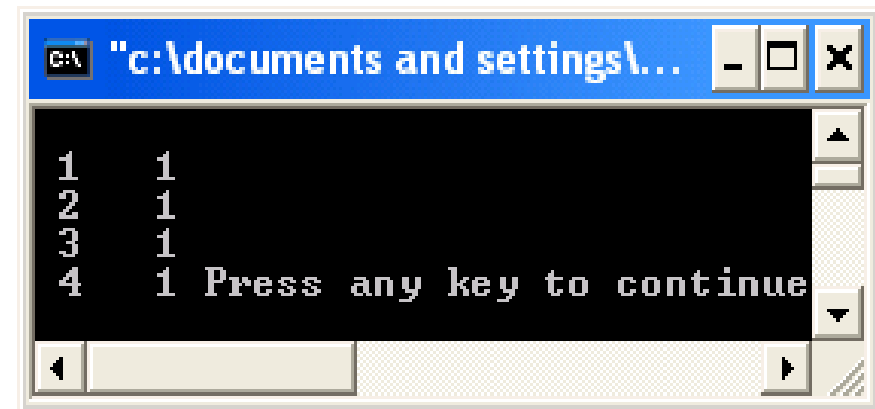
`int fsoma(void)`

`{`

`int soma=0;`

`return soma+=1;`

`}`



Exemplo 14

```
/* Qual o resultado da execução deste programa ? */

#include <stdio.h>

void func1(void);
void func2(void);
void func3(void);

int inic = 1;

void main(void)
{
    int inic = 99;
    int i;

    printf("\n Teste a func1():\n");
    for ( i=0 ; i<3 ; i++)
        func1( );

    printf("\n\n Em main(), inic = %d \n", inic);

    printf("\n Teste a func2():\n");
    for ( i=0 ; i<3 ; i++)
        func2( );
```

```

printf("\n\n Em main(), inic = %d \n", inic);

printf("\n Teste a func3():\n");
for ( i=0 ; i<3 ; i++)
    func3( );

printf("\n\n Em main(), inic = %d \n", inic);
}

void func1(void)
{
    static int inic=1;
    if (inic) {
        printf("\n Inicializacao ");
        inic=0;
    }
    printf("\n Continua");
}

void func2(void)
{
    if (inic) {
        printf("\n Inicializacao ");
        inic=0;
    }
    printf("\n Continua");
}

```

```

void func3(void){
    int inic=1;
    if (inic) {
        printf("\n Inicializacao ");
        inic=0;
    }
    printf("\n Continua");
}

```

Resultado da execução:

```

Teste a func1():

Inicializacao
Continua
Continua
Continua

Em main(), inic = 99

Teste a func2():

Inicializacao
Continua
Continua
Continua

Em main(), inic = 99

Teste a func3():

Inicializacao
Continua
Inicializacao
Continua
Inicializacao
Continua

Em main(), inic = 99
Press any key to continue_

```