Capítulo 2

Funções

Até o presente momento estudamos programas escritos de maneira estrutural e em um único bloco de código - inteiramente na função *main()*.

Esta técnica se mostra muito eficaz nos quesitos *rapidez de implementação* e *tempo de execução*. Contudo, este modelo é pouco útil no que diz respeito ao reaproveitamento de código.

Grande parte dos programas podem ser divididos em pequenos módulos chamados de **funções**. Esta técnica é conhecida pela filosofia do "Dividir para Conquistar". A combinação de várias pequenas funções forma, geralmente, um programa grande e complexo. As bibliotecas da linguagem C por exemplo, fornecem um grande número de funções prontas para o uso, tais como: *pow(), printf(), scanf(), malloc()*.

Existem outros tipos de funções na linguagem C, são aquelas definidas pelo programador e são conhecidas como **funções definidas**.

Formalmente, uma função é um bloco de código que se propõe a fazer uma tarefa bem definida dentro de um programa maior e, que possa ser reutilizada quantas vezes for necessário.

Existem alguns bons motivos para modularizar um programa em C, entre eles destacam-se:

- Flexibilização de um programa;
- Reutilização de código (OO);
- Evitar o retrabalho;

O conjunto de uma função principal com sub-funções (também chamadas de sub-rotinas) forma uma "hierarquia de código". A figura 2.1 mostra um exemplo da estrutura hierárquica que é criada pela linguagem C ao utilizar funções.

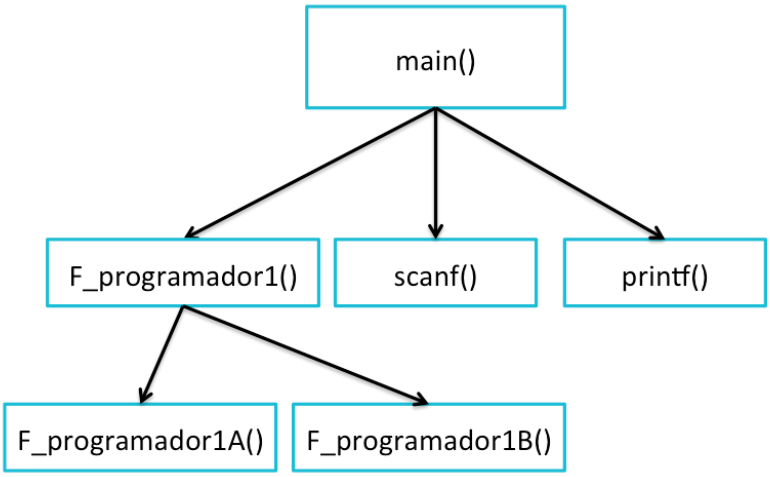

Figura 2.1: Hierarquia de Funções

2.1 Chamadas de Funções

As funções são ativadas (**chamadas** ou **invocadas**) por uma **chamada de função**. A chamada de função especifica o **nome da função** e fornece os **argumentos (parâmetros)** necessários (caso existam).

Sob a ótica da execução de um programa, a chamada de uma função segue as diretrizes do sistema. Para exemplificar, considere a função principal de qualquer programa em C (*main()*). Ao executar seu programa após compilá-lo, o seu terminal fica inutilizável para qualquer outra operação que não seja a execução do programa em si. Dizemos que o terminal está "suspenso"até que o programa encerre sua execução.

Isto ocorre porque o seu terminal é "a função"que chamou o seu programa (chamou *main()*). Do mesmo modo, quando você utiliza funções dentro do *main()*, o seu programa fica suspenso até que a execução da função (ou sub-rotina) seja encerrada.

A figura 2.2 mostra um exemplo de uma chamada de uma função dentro do main.

Vale salientar que este exemplo ilustra a função principal chamando uma sub-rotina. Contudo, uma sub-rotina também pode invocar outra função. Um bom exemplo é quando queremos mostrar o resultado de uma função matemática, raiz quadrada por exemplo, diretamente na função de saída (printf()). O comando seria este: printf("Raiz quadrada de 900: %.2f", sqrt(900.0));

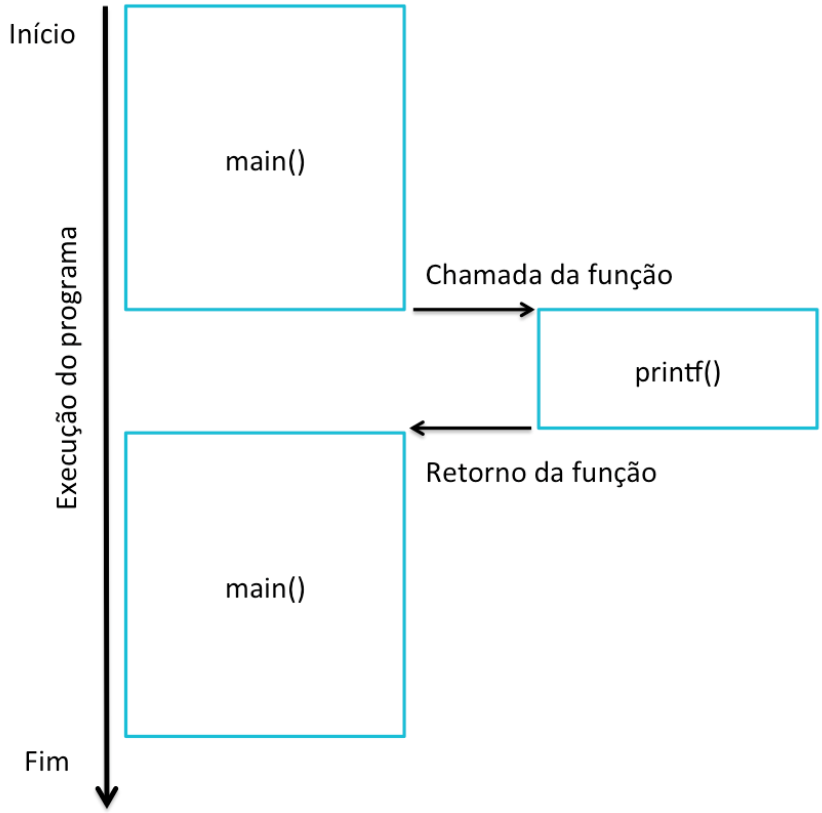

Figura 2.2: Chamada de Função

2.2 Parâmetros de Funções

As funções podem ou não receber parâmetros (também chamados de **argumentos**) ao serem invocadas pelo programa principal. Existem, basicamente, duas formas de passar argumentos às funções na linguagem C:

- Chamada por valor;
- Chamada por referência;

Tecnicamente falando existe apenas uma: chamada por valor. Acontece que com a utilização de ponteiros a linguagem C permite uma simulação de chamada por referência (a chamada por referência estudaremos adiante após os estudos sobre ponteiros).

A chamada por valor é feita pelo programa quando este copia o conteúdo da variável da função chamadora para a variável correspondente na função chamada. Este processo pode ser feito para um ou mais argumentos. Isto é uma característica importante da linguagem C. As funções podem receber vários argumentos como parâmetro.

2.3 Retorno de Função

Após realizarem suas tarefas as funções devem sinalizar para a função chamadora que finalizaram com sucesso (ou não) suas atividades. Pense no "return 0;"da função *main()*. Ele indica para quem o chamou (o terminal, ou o S.O.) que o programa finalizou corretamente. O número zero pode ou não ser utilizado para a continuidade da função chamadora. Neste caso é apenas um indicativo que tudo acabou bem.

Da mesma forma, uma função que foi projetada para somar dois números deve, ao final de sua execução, retornar a soma destes dois números. Do contrário perde-se o sentido de existir a função soma().

2.4 Variáveis Globais e Locais

Na programação estrutural pura, uma variável é declarada logo após o *main()* e serve para todo o programa. Quando um programa é modularizado ocorre uma sistematização sobre a utilização das variáveis. Entra em cena neste momento as variáveis globais.

Uma variável global é declarada fora da função *main()* e serve para todas as funções do programa. As variáveis declaradas dentro da função principal são chamadas agora de variáveis locais pois pertencem apenas a função *main()*. Se o programador desejar passar o valor de uma variável local em main ele deve fazer isto por meio de um argumento que será fornecido como parâmetro para a função a ser chamada.

Um exemplo deste conceito pode ser visto ao utilizar o printf() para exibir o valor de uma variável. A função printf() não conhece a variável local até que esta seja informada por meio de um argumento que é fornecido ao chamar o printf(). Ex.:

```
1 #include <stdio.h>
2 int main(){
3         int a = 5;
4         printf("O valor de a: %i", a);
5 return 0;
6 }
```

Uma variável global por sua vez pode ser acessada por qualquer função, indiferentemente de pertencer ou não ao *main()*. Estudaremos melhor este conceito no decorrer do capítulo.

2.5 Funções: Utilização e Conceitos

O que é melhor do que um bom exemplo para fixarmos os conceitos? Vamos consolidar os conceitos vistos até aqui com o código exemplo apresentado a seguir.

```
1  #include<stdio.h>
2
3  //prototipo das funcoes
4  int soma();
5  quadrado(int);
6
7  //variaveis globais
8  int v1, v2, res;
9
10 int main(){
11
12         int total, v3;
13
14         v1 = 5; //altera o conteúdo da variável global,
              vale para todas as funções
```

```
15          v2 = 3;
16          v3 = v1;
17
18          total = soma(); //chamando funcao sem passar
               argumentos mas esperando um retorno
19          quadrado(v3); //chamando funcao com argumentos
               mas sem retorno
20
21          printf("\n\t Soma de v1 com v2: %i ", total);
22          printf("\n\t Quadrado de v3: %i", res);
23
24  return 0;
25  }
26
27  int soma(){
28
29          int soma; // variável local, vale somente na
               funcao soma()
30          soma = v1 + v2; // soma as 2 variáveis globais
31          return soma; // retorna para a funcao chamadora
32  }
33
34  quadrado(int a){
35
36          res = a * a; // variavel global 'res' recebe o
               quadrado de 'a'
37  }
```

Este exemplo abrange todos os conceitos apresentados até aqui, vamos analisá-lo.

2.5.1 Protótipo da Função

As linhas 4 e 5 apresentam um **protótipo de função**. Note que cada função tem o seu protótipo. O protótipo de uma função é utilizado pelo compilador para checar se o uso da função é realizado corretamente pelo programa. Isto é, se a **quantidade, tipo e ordem** dos argumentos é passada corretamente durante a chamada da função. O protótipo deve ser declarado sempre antes da função *main()*.

Neste exemplo temos dois protótipos. O primeiro é da função soma(). Este protótipo indica que a função irá retornar um **inteiro** (int) e não espera receber nenhum argumento. O segundo protótipo é da função quadrado(). Este protótipo indica que a função não irá retornar nada. Contudo, ela espera

receber como parâmetro um **valor do tipo inteiro**.

Perceba que ao utilizar variáveis globais o programa pode ter um misto entre funções que retornam e que não retornam valores nenhum à função chamadora. Neste caso a função soma() retorna a soma dos valores v1 e v2 que são globais, portanto são acessíveis a partir de qualquer função, para uma variável que está aguardando (ansiosa) na função chamadora (main()) e, que por sua vez pertence somente à função principal.

Da mesma forma o entendimento é válido para os **parâmetros**. Por utilizar variáveis globais, o programa permite que uma determinada função execute tarefas sobre estas variáveis sem as possuir na própria função. Em contrapartida às variáveis globais, as funções podem receber uma cópia das variáveis locais da função chamadora por meio dos parâmetros.

Isto pode ser visto no exemplo a partir da função quadrado(), ela recebe um argumento do tipo inteiro e irá receber uma cópia do valor contido na variável (v3) que está em main(). Essa cópia é manipulada na função e seu valor só é válido na própria função. Entretanto, o resultado produzido é armazenado em uma variável global, logo, ela está disponível para qualquer outra função, inclusive o main().

2.5.2 Definição da Função

A definição de uma função obedece a seguinte sintaxe:

```
1  tipo_do_valor_de_retorno NOME_DA_FUNCAO(
       lista_de_parametros)
2  {
3          declarações;
4          instruções;
5          retorno;
6  }
```

Onde:

- **tipo_do_valor_de_retorno**: Indica o tipo de dado do resultado devolvido à função chamadora. Se nada for especificado, o tipo **int** é assumido. Neste exemplo temos duas funções com retorno do tipo inteiro. A função soma() apresenta de forma explícita que irá retornar um tipo inteiro. Já a função quadrado() assume um tipo inteiro como retorno por definição, uma vez que não possui tipo explícito de retorno.

- NOME_DA_FUNCAO(): O nome da função segue os critérios estudados sobre identificadores da linguagem C. Isto é, as regras válidas para variáveis também são aplicadas aos nomes de funções.

Uma dica importante em relação ao nome da função é que este deve ser sucinto e expressar efetivamente a tarefa da função. Se você não conseguir dar um nome simples e pequeno que resuma o que a função faz é porque ela faz coisas demais e talvez precise ser dividida em funções menores. Um bom exemplo é: tirar_media_e_imprimir(); Talvez esta função deva ser dividida em duas menores: obter_media(); imprimir_resultado(); Assim, elas se limitam a fazer o que seu nome sugere e podem facilmente serem reaproveitadas em outros trechos do programa.

- *lista_de_parametros*: É uma lista separada por vírgulas contendo as declarações dos argumentos esperados quando a função é chamada. Se nada for especificado o padrão é **void**. Se houver parâmetros mas não tipos, **int** é assumido. Em nosso exemplo temos uma função que não espera nada como argumento (função soma()) e outra que espera receber um dado do tipo inteiro (quadrado(int)).

- **declarações; instruções; retorno;**: Esta sequência de itens é equivalente para todas as funções e seguem a ideia aplicada até o presente momento onde havia apenas o *main()*. Isto é, declara-se as variáveis locais, utiliza-se elas nas instruções e retorna-se um valor esperado (quando houver).

2.5.3 Chamada da Função

Podemos notar nas linhas 17 e 18 as chamadas de funções aplicadas sobre as funções soma() e quadrado(). Esta é a etapa para o qual as funções são projetadas: a utilização e reutilização. Neste momento o programador deve tomar o cuidado de invocar as funções obedecendo os critérios estabelecidos nos protótipos. Isto é, na função soma() deve-se chamá-la sem parâmetro algum porém, aguardando o retorno da função (com a variável "total"). Já na função quadrado() deve-se passar o argumento (variável v3, um **valor do tipo inteiro**) conforme especificado no protótipo e não aguardar nenhum retorno.

2.6 Resumo

Um resumo sobre o passo-a-passo de como utilizar funções em C pode ser elencado nos seguintes itens:

- Declarar o Protótipo da Função

- Definir a Função
- Invocar (chamar) a Função no programa

2.7 Exemplos

Alguns exemplos de programas modularizados por meio de funções.

Exemplo 1

```
1  #include <stdio.h>
2
3  //prototipo da funcao
4  int numeros_iguais(int, int);
5
6  int main(){
7
8          int a, b, resposta;
9
10         printf("\n\tDigite 2 valores inteiros: ");
11         scanf("%d%d", &a, &b);
12
13         resposta = numeros_iguais(a,b);
14         if(resposta == 1)
15                 printf("\n\tOs numeros fornecidos sao
                      iguais.");
16         else
17                 printf("\n\tOs numeros fornecidos sao
                      diferentes.");
18
19 printf("\n\n");
20 return 0;
21 }
22 //definicao da funcao
23 int numeros_iguais(int n1, int n2){
24
25         if(n1 == n2)
26                 return 1;
27         else
28                 return 0;
29
30 }
```

Exemplo 2

```c
#include <stdio.h>

//prototipo da funcao
int maior_numero(int, int);

int main(){

        int a, b, resposta;

        printf("\n\tDigite 2 valores inteiros: ");
        scanf("%d%d", &a, &b);

        resposta = maior_numero(a,b);
        printf("\n\tO maior numero eh: %d", resposta);

printf("\n\n");
return 0;
}
//definicao da funcao
int maior_numero(int n1, int n2){

        if(n1 >= n2)
                return n1;
        else
                return n2;

}
```

Exemplo 3

```c
#include <stdio.h>

//prototipo da funcao
char converte_maiusculo(char);

int main(){

        char min, mai;

        printf("\n\tDigite uma letra: ");
        scanf(" %c", &min);

        mai = converte_maiusculo(min);
        if(mai != '*')
```

```
15                  printf("\n\tLetra %c convertida para
                       maiusculo: %c", min, mai);
16          else
17                  printf("\n\tLetra %c ja esta em maiusculo
                       .", min);
18
19  printf("\n\n");
20  return 0;
21  }
22  //definicao da funcao
23  char converte_maiusculo(char l){
24
25          char M;
26
27          if(l >= 97 && l <= 122)
28                  M = l - 32;
29          else
30                  M = '*';
31
32          return M;
33  }
```

2.8 Recursividade

Este assunto é amplamente discutido em Deitel [3] no capítulo 5 e em Damas [1] no capítulo 9. Este material traz o tema de forma sucinta apenas a título de conhecimento. Portanto, para aprofundar-se nos estudos procure e leia os capítulos dos livros indicados.

O conceito de recursividade em funções diz respeito à capacidade que uma função tem de chamar a si mesmo. A ideia que norteia as funções recursivas é de que uma função resolve um problema básico (bem básico). Deste modo, recursividade pode ser aplicado à problemas que podem ser divididos em duas partes teóricas: a parte que a função SABE resolver e a parte que a função NÃO SABE resolver. Para ser viável, a parte que a função não sabe resolver deve ser semelhante a parte que a função sabe resolver. Então, ela divide o problema e para a parte que não sabe resolver ela chama a si mesma.

Um bom exemplo é o problema matemático conhecido como fatorial. O fatorial de um inteiro não-negativo a, escrito n! (e pronunciado "fatorial de n"), é o produto

```
1  n * (n - 1) * (n - 2) * ... * 1
```

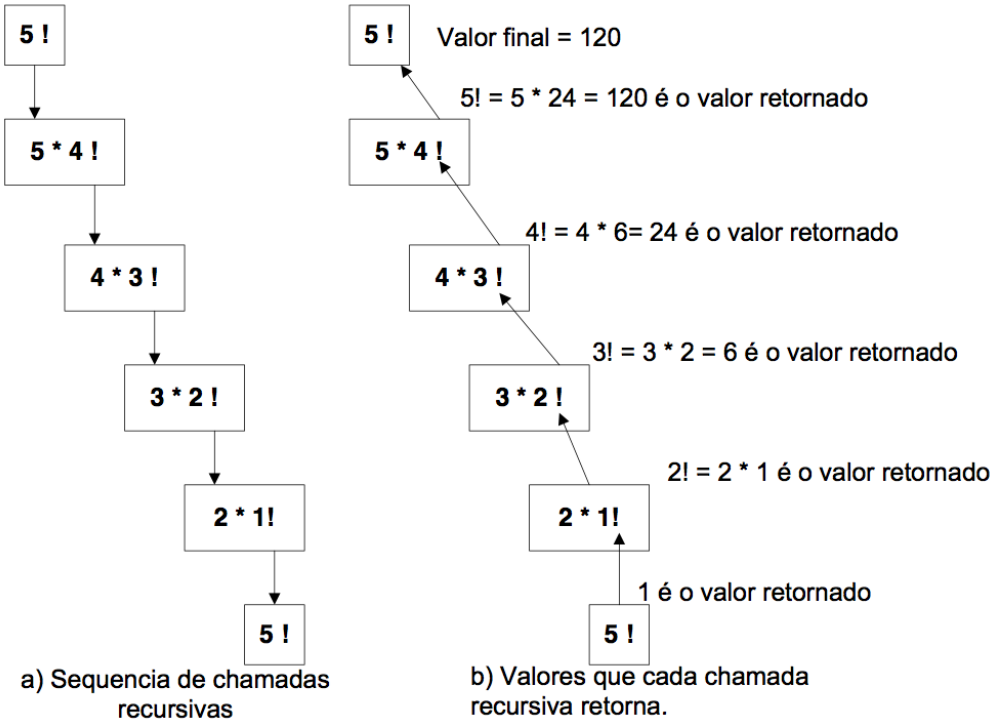

Figura 2.3: Função recursiva para cálculo do fatorial

com 1! igual a 1 e definindo 0! igual a 1. Por exemplo, 5! é o produto 5*4*3*2*1, que é igual a 120 O fatorial de um inteiro, numero, maior ou igual a 0, pode ser calculado iterativamente (não-recursivamente) usando for como se segue:

```
1 fatorial = 1;
2 for(contador = numero; contador >= 1; contador--)
     fatorial *= contador;
```

Chega-se a uma definição recursiva para a função fatorial observando o seguinte relacionamento:

```
1 n! = n * (n - 1)!
```

Por exemplo, 5! é claramente igual a 5 * 4! como é mostrado a seguir: 5! = 5 * 4 * 3 * 2 * 1 5! = 5 * (4 * 3 * 2 * 1) 5! = 5 * (4!).

A função recursiva fatorial verifica inicialmente se uma condição de término é verdadeira, i.e., se numero é menor ou igual a 1. Se numero for menor ou igual a 1, fatorial retorna 1, nenhuma recursão se faz mais necessária e o programa é encerrado. Se numero for maior do que 1, a instrução

return numero * fatorial (numero - 1);

expressa o problema como o produto de numero por uma chamada recursiva a fatorial calculando o fatorial de numero - 1. Observe que fatorial (numero - 1) é um problema ligeiramente mais simples do que o cálculo original fatorial (numero).

O código-fonte para o exemplo da figura 2.3 pode ser visto a seguir:

```
1  #include <stdio.h>
2
3  int fatorial(int);
4
5  int main(){
6
7          int numero, resultado;
8          printf("\n\tForneca um numero para fatorar: ");
9          scanf("%d", &numero);
10
11         resultado = fatorial(numero);
12
13         printf("\n\tO fatorial de %d eh: %d", numero,
               resultado);
14
15 printf("\n\n");
16 return 0;
17 }
18
19 int fatorial(int n){
20
21         if((n == 1) || (n == 0))
22                 return 1;
23         else
24                 return (fatorial(n - 1) * n);
25
26 }
```

2.9 Biblioteca Math.h

A linguagem C oferece por meio da biblioteca <math.h> uma série de funções matemáticas prontas para uso. Elas são muito úteis quando desejamos, rapidamente, resolver uma equação matemática sem precisar gastar um bom tempo montando sua própria função.

O código exemplo a seguir contém boa parte das funções. Os comentários tornam o código auto-explicativo dispensando desta forma, longos (e

cansativos) textos. Para uma compreensão otimizada sobre a biblioteca, execute este código em sua máquina procurando alterar os valores utilizados nas funções.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5
6  int main (void)
7  {
8          double x = 9.75;
9
10         printf("\n\t=== Biblioteca math.h ===");
11         printf("\n\tFuncoes Basicas (arredondamento):");
12
13         printf("\n\tValor original de x = %f", x);
14         printf("\n\tValor aproximado para baixo %f \n",
               floor(x));
15         printf("\tValor aproximado para cima %f \n", ceil
               (x));
16
17         printf("\n\t=== === === === === ===\n");
18
19         printf("\n\tFuncoes de raiz e potenciacao \n\n");
20         printf("\tValor original de x = %lf\n",x);
21         printf("\tValor da raiz quadrada %f \n", sqrt(x))
               ;
22
23         x = ceil(x); //arredondando o x para cima, x
               passa a valer 10
24         printf("\tValor de %.2lf ao quadrado %.2f \n", x,
                pow(x,2));
25
26         printf("\n\t=== === === === === ===\n");
27
28         printf("\n\tFuncoes Trigonometricas\n\n");
29         x = 0; //atribuindo zero em x para fazer os
               cálculos trigonométricos
30
31         printf("\tValor de  seno de %.2f = %.2f \n", x,
               sin(x));
32         printf("\tValor de  cosseno de %.2f = %.2f \n", x
               , cos(x));
```

```
33          printf("\tValor de  tangente de %.2f = %.2f \n\n"
               , x, tan(x));
34
35          printf("\n\t=== === === === === ===\n");
36
37          printf("\n\tFuncoes logaritmicas\n\n");
38
39          x = 2.718282;
40          printf("\tLogaritmo natural de x %.2f = %.2f \n",
                x, log(x));
41
42          x = 10;
43          printf("\tLogaritmo de x na base 10 %.2f = %.2f \
               n", x, log10(x));
44
45  printf("\n\n");
46  return 0;
47  }
```

Nota: para compilar programas que utilizam a biblioteca <math.h> é necessário incluir o parâmetro **-lm** na linha de comando, desta forma:

gcc -lm fonte.c -o binario