Modularização: Funções em C

Notas de Aula

Prof. Francisco Rapchan
www.geocities.com/chicorapchan
rapchan@terra.com.br

Muitas vezes um problema grande pode ser resolvido mais facilmente se for dividido em pequenas partes. Tratar partes menores e mais simples em separado é muito mais fácil do que tratar o problema grande e difícil de uma vez.

O que vimos da linguagem C até agora é adequado para escrevermos pequenos programas. Entretanto, se for necessário desenvolver e testar programas mais sofisticados (que envolvam mais funcionalidades) deveremos usar técnicas que nos permitam, de alguma forma, organizar o código fonte. Alguns aspectos devem ser levados em consideração:

- Programas complexos são compostos por um conjunto de segmentos de código não tão complexos.
- Muitas vezes usamos em um programa trechos que já desenvolvemos em outros programas.

É comum em programação decompor programas complexos em programas menores e depois juntá-los para compor o programa final. Essa técnica de programação é denominada **programação modular**.

A programação modular facilita a construção de programas grandes e complexos, através de sua divisão em pequenos módulos, ou subprogramas, mais simples. Estes subprogramas alem de serem mais simples de serem construídos, são também mais simples de serem testados. Esta técnica também possibilita o reaproveitamento de código, pois podemos utilizar um módulo quantas vezes for necessário eliminando a necessidade de escrevê-lo repetidamente.

Outro aspecto importante da modularização é a possibilidade de vários programadores trabalhem simultaneamente na solução de um mesmo problema, através da codificação separada dos módulos. Assim, cada equipe pode trabalhar em um certo conjunto de módulos ou subprogramas. Posteriormente estes módulos são integrados formando o programa final. Em outras palavras, quando for necessário construir um programa grande, devemos dividi-lo em partes e então desenvolver e testar cada parte separadamente. Mais tarde, tais partes serão acopladas para formar o programa completo.

A modularização, em C começa através do uso adequado de *funções (**functions**)*. Funções são algumas das formas usadas para agruparmos código de forma organizada e modularizada. Tipicamente, usamos funções para realizar tarefas que se repetem várias vezes na execução de um mesmo programa.

Isso é feito associando-se um nome a uma seqüência de comandos através do que chamamos de *Declaração da Função*. Pode-se então usar o nome do procedimento ou da função dentro do corpo do programa, sempre que desejarmos que o seu bloco de comandos seja executado, isso é o que chamamos de *Chamada do Procedimento ou da Função*.

Funções básicas

Na matemática fazemos uso intenso de funções. Por exemplo, podemos definir as funções:

- Função quadrado: $f(x) = x^2$ Exemplo: f(2) = 4
- Função reverso: $f(x) = 1/x$ Exemplo: f(2) = 0,5
- Função dobro: $f(x) = x * 2$ Exemplo: f(3) = 6

Estas funções possuem apenas um parâmetro. Podemos também definir funções com mais de um parâmetro:

- Função soma dois números: $f(x, y) = x + y$
- Função hipotenusa: $f(x, y) = \sqrt{x + y}$

  O resultado desta função para **x = 3** e **y = 4** é **5**.
  Portanto, f(3,4) da função hipotenusa **retorna** 5.
  Dizemos também que x e y são os **parâmetros da função**.
  Os valores 3 e 4 são os **argumentos** da função hipotenusa.

Nas linguagens de programação também temos funções e elas são bem parecidas com as funções da matemática. Uma função é um tipo especial de sub-rotina que retorna um resultado de volta ao "ponto" onde foi chamada.

Exemplo 1. Construa uma função para somar dois números conforme definida acima.

```
#include <stdio.h>

// Declara a função
int soma (int x, int y)
{
   int s;
   s = x + y;
   return (s);
}

int main(void)
{  int c;

   // Usa a função soma
   c = soma (3 , 5);

   // Mostra o resultado
   printf ("Resultado: %i\n",c);

   // Retorno da função principal
   return 0;
}
```

Neste programa definimos uma função chamada **soma**.

Embora chamar uma função de **f** ou **g** seja prática comum na matemática, na informática normalmente são usados nomes mais expressivos ou significativos. Uma dica: os nomes usados devem fazer menção ao uso ou ao comportamento da função.

Observe que definimos a função soma **fora da função main**. Em C, main ( ) também é uma função. É a primeira função que é executada.

Nesse programa, os parâmetros da função soma são **x** (do tipo int) e **y** (também do tipo int) e o tipo de retorno da função, coincidentemente, também int.

No corpo da função os parâmetros são usados como variáveis comuns.

Outro elemento importante é o **return**. Ele serve para indicar qual o valor que a função irá retornar.

No corpo do programa, chamamos a função soma, passando como **argumento** os valores 3 e 5.

Observe que a partir desse exemplo, colocamos a função **main como sendo do tipo int**. Na verdade esse fato está apenas sendo explicitado, pois quando não damos um tipo para a função main, o compilador assume o tipo int.

A mesma coisa acontece com o **return** da função main: se não o declaramos, o compilador inclui um return 0.

Como estamos usando a função main sem nenhum argumento, colocamos main (void). Isso explicita que main não terá nenhum argumento. Declarar apenas main ( ) indica que a função main pode ter um número qualquer de argumentos.

A função main obrigatoriamente deve ter um valor inteiro como retorno. Esse valor pode ser usado pelo sistema operacional para testar a execução do programa. A convenção geralmente utilizada faz com que a função main retorne zero no caso da execução ser bem sucedida ou diferente de zero no caso de problemas durante a execução.

Exemplo 2. O programa abaixo mostra o valor do fatorial dos números 1 até 10.

```
#include <stdio.h>

long int fatorial (int n)
{
   int i;
   long int fat;

   fat = 1;

   for (i=1; i <= n; i++)
      fat = fat * i;

   return fat;
}

main(void)
{
   int i;
   for (i = 1; i <= 100; i++)
      printf("%i! = %li\n",i,fatorial(i));
}
```

Note que temos duas variáveis com nome i definidas neste programa. Uma é definida para o programa principal e outra para a função.

O compilador trata ambas separadamente.

Dizemos que o escopo das duas variáveis é diferente. Uma tem como escopo o programa principal e outra tem como escopo apenas a função, ou seja, ela é enxergada apenas dentro da função.

A saída deste programa será:

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Observe também que declaramos o retorno da função fatorial e a variável fat como **long int**. A linguagem C oferece uma série de tipos que permite trabalhar com números de diferentes precisões. Veja a tabela abaixo:

Tipo	Bytes	Formato	Inicio	Fim
char	1	%c	-128	127
unsigned char	1	%c	0	255
int	2	%i	-32.768	32.767
unsigned int	2	%u	0	65.535
long int	4	%li	-2.147.483.648	2.147.483.647
unsigned long int	4	%lu	0	4.294.967.295
float	4	%f	3,4E-38	3.4E+38
double	8	%lf	1,7E-308	1,7E+308
long double	10	%Lf	3,4E-4932	3,4E+4932

Escopo de variáveis

Escopo de uma variável refere-se ao âmbito em que ela pode ser usada. Uma variável definida dentro de uma função só pode ser usada dentro desta função (não pode ser usada fora do escopo da função). Portanto, podemos definir variáveis com o mesmo nome em funções diferentes sem nenhum problema. Dizemos que as variáveis que são definidas dentro de uma função **são locais** a essa função, ou seja, elas só existem enquanto a função está sendo executada (elas passam a existir quando ocorre a entrada da função e são destruídas ao sair).

Um aspecto importante é que **em C não é possível definir uma função dentro de uma outra função**.

Exemplo 3. Observe o programa abaixo.

```
#include <stdio.h>

// Variáveis globais
int s, a;

int soma (int i)
{
   int v;

   v = s + i;
   s = v;
   return (v);
}

int main(void)
{
   s = 1;
   a = soma (5);
   printf ("A: %i  S: %i\n",a,s);
   return 0;
}
```

Neste programa, a função **soma** usa 3 variávies:

- **i**: definida como parâmetro da função.
- **v**: definida como variável da função.
- **s**: definida como variável global do programa.

Observa que as variáveis **s** e **a** também são enxergadas de dentro da função soma. Desta forma, os vares das **variáveis globais** (definidas para todo o programa) podem ser usadas ou alteradas de dentro de uma função.

Por outro lado, vemos que há duas **variáveis locais** na função soma: **i** e **v**. Estas duas variáveis só são enxergadas pelo código da própria função **soma.** Não é possível à outras funções acessar o valor destas variáveis.

Dizemos que as variáveis definidas para o programa todo são **globais** (possuem escopo ou âmbito global) em relação às funções e que as variáveis definidas dentro das funções (inclusive os parâmetros) são **locais** (possuem escopo ou âmbito local) à função.

Uma pergunta comum com relação a variáveis globais e locais é a seguinte: e se for definida uma variável local com o mesmo nome de uma variável global? Neste caso teremos duas variáveis completamente diferentes. Localmente não será possível acessar a variável global e vice–versa.

Observação: De forma geral **deve-se evitar o uso de variáveis globais**. O uso de variáveis globais é considerada má prática de programação pois torna o código muito dependente, aumentando o acoplamento da função e tornando-a difícil de ser usada em outro programa, diminuindo assim sua portabilidade.

Procedimentos (funções que retornam void – vazio)

Chamamos de **procedimentos** as funções que não retornam nenhum valor.

Exemplo 4. O programa abaixo incrementa o valor da variável global **s**.

```
#include <stdio.h>

int s;

void soma (int i)
{
   s = s + i;
}

int main(void)
{
   s = 1;
   soma (5);
   printf ("S: %i\n",s);
   return 0;
}
```

Este é um programa muito simples que serve apenas para mostrar como usar procedimentos.

Observe que foi definida a função soma como retornando void. O termo void significa vazio ou sem valor. Assim, chamamos funções desse tipo com sendo procedimentos. Como qualquer procedimento, este **não retorna nenhum valor**.

Neste caso o procedimento soma é chamado com o argumento 5 e altera o valor da variável global s.

O programa se comporta como se na linha em que o procedimento é chamado, o **fluxo de execução fosse desviado** para o procedimento e depois retornasse para a instrução imediatamente seguinte ao procedimento.

O procedimento funciona como uma **sub-rotina**.

Atenção: a função main é uma função especial. Não é possível declarar main como void. Assim, a definição abaixo está conceitualmente errada (embora em alguns compiladores possa passar desapercebido):

```
void main ( ) // Errado !!!
```

Na verdade, o único tipo válido para a função main é int.

```
int main (void) // Certo!!!
```

Exemplo 5. Não é possível fazer as operações abaixo:

```
#include <stdio.h>

int s,a;

void soma (int i)
{
   s = s + i;
}

int main(void)
{
   s = 1;
   a = soma (5); // Errado!!!
   printf ("A: %i\n",a);
   return 0;
}
```

Observe que um procedimento **não retorna valor.**

Portanto, não podemos fazer atribuição de um procedimento a uma variável nem passá-lo como argumento.

Passagem de Parâmetro por Referência (ponteiros)

Há duas formas de passagem de parâmetros:

- Passagem de parâmetros por valor.
- Passagem de parâmetros por referência.

A **passagem de parâmetros por valor** é a forma que temos usado em todos os exemplos até agora. Dizemos que parâmetros passados por valor são parâmetros de entrada. O valor do argumento é passado para dentro da função através dos parâmetros. Assim, os parâmetros recebem os valores dos argumentos.

Exemplo 6. O código apresentado abaixo usa o procedimento **soma** para incrementar o valor de **s** em **i** unidades.

```
#include<stdio.h>
int a,s;

void soma (int i)
{
   s = s + i;
}

int main (void)
{  s = 1;
   a = 1;
   soma (5);
   printf ("%i  %i\n",a,s);
}
```

No procedimento **soma,** o parâmetro **i** é passado "por valor". Dizemos que **i** é um parâmetro de entrada.

Quando o programa executa o procedimento **soma(5)** o argumento 5 é passado para o parâmetro **i** que irá armazenar este valor.

O parâmetro i é uma variável que ocupa um espaço em memória e, neste caso, coloca o valor 5 neste espaço.

Observe que o procedimento **soma** é muito pouco versátil. Ele só pode incrementar o valor de **s**. Caso quiséssemos alterar o valor de **a** teríamos que criar um outro procedimento **soma** para **a**.

Seria interessante um procedimento em que tivéssemos dois parâmetros: um indicando a variável que desejamos alterar e outro o valor que queremos somar!

Mas não seria tão fácil. O código abaixo, por exemplo, não funcionaria adequadamente:

```
#include<stdio.h>

int a,s;  // Variáveis globais

void soma (int n, int i)
{
   n = n + i;
}

int main (void)
{  s = 1;
   a = 1;
   soma (s,5);
   soma (a,3);
   printf ("%i  %i\n",a,s);
   system ("PAUSE");
}
```

Neste caso, o procedimento **soma** é chamado duas vezes. Na primeira o parâmetro **n** recebe o argumento **s** que vale 1. Então **n** passa a valer 1 e é somado com 5 dando 6 como resultado. Mas **n** não altera o valor de **s**. O parâmetro **n** existe apenas durante o breve tempo em que o procedimento **soma** é executado. Assim que o procedimento termina sua execução a área de memória ocupada por **n** é liberada (apagada). Desta forma, o resultado impresso será 1 1.

Na **passagem de parâmetro por referência** não é criado um novo espaço de memória para o parâmetro. O que acontece é que o parâmetro vai usar o mesmo espaço de memória usado pelo argumento. Nesse caso é passado como argumento um ponteiro para a variável e não o seu valor.

Para indicar que um parâmetro é passado por valor e não por referência, colocamos a expressão ***** na frente dele na declaração da função ou do procedimento.

Assim, corrigindo o exemplo dado acima:

```
#include<stdio.h>

int a,s;

void soma (int * n, int i)
{
   *n = *n + i;
}

int main (void)
{  s = 1;
   a = 1;
   soma (&s,5);
   soma (&a,3);
   printf ("%i  %i\n",a,s);
}
```

Neste caso o procedimento **soma** tem o parâmetro **n** passado por referência e o parâmetro **i** passado por valor. Dizemos que n é um ponteiro para um inteiro qualquer.

Na chamada do procedimento **soma(&s, 5)** o parâmetro **n** irá compartilhar a mesma área de memória da variável **s**, ou seja, será passado o endereço de s para o parâmetro n. Assim, alterar **n** equivale a alterar **s.** Na chamada seguinte, **soma(&a, 3)** o parâmetro **n** irá compartilhar a mesma área de memória da variável **a** (o endereço de a é passado como argumento). Assim, alterar **n** equivale a alterar **a.** O resultado impresso será 4 6.

Exemplo 7. Faça um procedimento que execute um swap, ou seja, que troque os valores de dois argumentos.

```
#include<stdio.h>

// Procedimento troca (swap)
void troca (int *a, int *b)
{
   int aux;
   aux = *a;
   *a  = *b;
   *b  = aux;
}

int main (void)
{  // Cria as variáveis e inicia seus valores
   int x = 2, y = 5;

   // Executa o prcedimento troca()
   troca (&x,&y);

   // Mostra os valores invertidos por troca()
   printf ("%i  %i\n",x,y);
}
```

O procedimento troca possui dois parâmetros: a e b. Ambos são do tipo inteiro e ambos são ponteiros para inteiros (ou seja, são **parâmetros por referência**).

Isso significa que o procedimento deverá receber os **ponteiros para as variáveis** passados como argumento.

No programa principal, troca é chamado passando como argumento as variáveis x e y. Lembre-se que, na verdade, são passados para troca os **ponteiros** para x e y!

Assim, qualquer alteração que troca fizer em a ou b será como se estivesse alterando diretamente x e y.

Vamos descrever, passo a passo, o que está sendo feito na função troca:

```
aux = *a;  //aux está recebendo o valor apontado por a
*a  = *b;  //o valor apontado por a está sendo substituído pelo o valor apontado por b
*b  = aux; //o valor apontado por b está sendo substituído pelo valor de aux
```

Podemos chamar uma função de dentro da própria função. Essa técnica é chamada de recursividade. Se uma função F chama a própria função F, dizemos que ocorre uma recursão. A implementação de alguns algoritmos fica muito mais fácil usando recursividade. Quando uma função é chamada recursivamente, cria-se um ambiente local para cada chamada. As variáveis locais de chamadas recursivas são independentes entre si, como se estivéssemos chamando funções diferentes.

Exemplo 8. O código apresentado abaixo usa o procedimento **soma** para incrementar o valor de **s** em **i** unidades.

```
#include <stdio.h>

/* Calcula o fatorial usando recursividade */
long int fat (int n)
{
   if (n==0)
      return 1;
   else
      return n*fat(n-1);
}

main (void)
{
   // Mostra o fatorial de 10
   printf ("%i\n", fat(10) );
}
```

Exemplo 9. A série de Fibonacci é muito conhecida na matemática. A série é composta assim, o primeiro e o segundo termos são 1. A partir do terceiro termo, todos os termos são a soma dos dois últimos.

Série de Fibonacci: 1, 1, 2, 3, 5, 8, 13, 21, 34...

Faça uma função que encontre o enésimo termo da seqüência de Fibonacci. Use recursividade.

```
#include <stdio.h>
int fib(int n)
{
   if (n>2)
      return ( fib(n-1) + fib(n-2) );
   else
      return 1;
}

main()
{
   int n;
   printf("Digite um numero: ");
   scanf("%d", &n);
   printf("O termo %d da serie de Fibonacci e: %d\n", n, fib(n));
}
```

Passando parâmetros para um programa (os argumentos argc e argv)

A função main() é uma função especial que pode receber dois parâmetros:

int main (int argc, char *argv[])

- argc (argument count) é um inteiro e possui o número de argumentos com os quais o programa foi chamado na linha de comando. Ele é no mínimo 1, pois o nome do programa é contado como sendo o primeiro argumento.
- argv (argument values) é um ponteiro para um vetor de strings . Cada string desse vetor é um dos parâmetros da linha de comando. argv[0] aponta para o nome do programa que é o primeiro argumento.

Os nomes dos parâmetros "argc" e "argv" podem ser mudados mas, por questão de padronização, costuma-se não modificá-los.

Exemplo 10. Faça um programa de forma que eu digite o nome do programa seguido de alguns números inteiros e o programa mostre para mim a soma desses números.

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

int main ( int argc, char * argv[] )
{
   int i, s;
   s = 0;
   for (i = 1; i <= argc ; i++)
      s = s + atoi ( argv [i] );
   printf ("Soma dos números: %i\n",s);
}
```

Supondo que esse programa se chame teste, ele poderia ser executado assim:

```
teste 2 8 5
Soma dos números: 15
```

Em argc temos o número de argumentos e no vetor argv temos a lista desses argumentos.

argv:

Teste	2	8	5

argc:

4

A função atoi converte um string em um inteiro e está na biblioteca stdlib.h.

Exemplo 11. Faça um programa em que o usuário digite o nome de um arquivo e seja mostrado o seu conteúdo.

```
#include <stdio.h>
int main ( int argc, char * argv[] )
{  FILE *arquivo;
   char c;

   // Abre o arquivo para leitura
   arquivo = fopen (argv[1], "r");
   if (arquivo == NULL){
      printf ("Erro ao abrir o arquivo:%s",argv[1]);
      return 1;
   }

   // Lê caracteres até o fim do arquivo
   while((c = getc(arquivo)) != EOF)
      printf("%c", c);

   fclose (arquivo);
   return 0;
}
```

Suponha que este programa chame-se mostra. Então poderíamos executá-lo assim:

> mostra teste.c

Neste caso, seria mostrado o conteúdo do arquivo teste.c na tela.

Observe também o comando

while ((c = getc (arquivo)) != EOF)

Nesse comando estamos usando getc para ler um caractere. Este caractere está sendo atribuído à variável c. Depois o conteúdo da variável c está sendo comparado a EOF (caractere de fim de arquivo). Assim, enquanto o caractere lido não for EOF, mostre na tela.

Poderíamos substituir o comando **while** deste programa pelo seguinte comando **for**:

for (c = getc(arquivo) **;** c != EOF **;** c = getc(arquivo) **)**

Exemplo 12. O programa abaixo mostra o texto o valor do fatorial dos números 1 até 10.

```
#include <stdio.h>

void mensagem (void)
{
   printf ("Ola Mundo!\n");
}

int main (void)
{
   mensagem();
   return 0;
}
```

Observe que a função **mensagem** não tem nenhum argumento (void) e também não retorna nenhum valor.

Funções que não retornam valor também são chamadas de **procedimentos**.

Exemplo 13. O programa abaixo calcula a média de um conjunto de elementos.

```
#include <stdio.h>

#define MAX  10  //define MAX com valor 10

// protótipo da função
float media (int vet_num[], int limite);

// função principal
int main (void)
{
   int num, i, vet[MAX];

   printf ("Número de elementos: ");
   scanf ("%d",&num);

   for (i = 0; i< num; i++) {
      printf ("Número %d: ",i+1);
      scanf ("%d",&vet[i]);
   }
   printf ("Média: %f\n", media (vet, num));
   return 0;
}

// definição da função
float media (int vet_num[], int limite)
{
   int i, soma = 0;

   for (i = 0; i< limite; i++)
      soma = soma + vet_num[i];

   return (soma / limite);
}
```

A expressão **#define** permite definir algumas **macros de compilação**. Neste programa, onde houver o termo MAX, será substituído por 10. Esta substituição é feita pelo próprio compilador imediatamente antes de compilar.

Outra coisa interessante neste programa é que mostra o uso de **protótipos de função**. A idéia é que é permitido usar uma função antes de defini-la desde que tenha sido definido pelo menos o seu protótipo, ou seja: o seu tipo de retorno, seu nome e os tipos de seus argumentos.

Observe também que, ao contrário das variáveis comuns, o conteúdo de um vetor **pode ser modificado** pela função chamada. Isto ocorre porque a passagem de **vetores** para funções é feita *por referência*.