TÉCNICAS DE PROGRAMAÇÃO

UNIDADE 2 - FUNÇÕES

Fernando Cortez Sica

Introdução

Iniciamos aqui mais uma unidade de nossa conversa sobre técnicas de programação. Agora falaremos sobre funções. A primeira imagem que pode vir à sua cabeça é uma função matemática "*f(x)*": falaremos de matemática? Não propriamente dito, falaremos sobre uma outra forma de estruturar um código de forma a definir um bloco de instruções para uma certa funcionalidade. Mas, então, funções servem apenas para deixar o programa estruturado, mais fácil de ser visualizado? Sim, um dos objetivos é este, mas há, também, outros motivos para elas existirem, é o que veremos nessa unidade. Sabemos que uma função encapsula um conjunto de instruções. O programa principal "**main()**" é uma função? Exato! O programa principal "**main()**" foi a primeira função que manipulamos aqui na nossa disciplina. Aqui falaremos também sobre outras, mas, principalmente, falaremos de como criar funções. Mas, se função é um bloco de instruções para uma certa funcionalidade, então, os blocos delimitados por "{" e "}" dos comandos condicionais e comandos de repetição são também funções? Não, funções são mais abrangentes: elas envolvem as instruções de forma mais ampla, ou seja, envolvem, inclusive, os comandos condicionais e os laços de repetição.
Neste capítulo abordaremos os aspectos relacionados aos conceitos e implementação de funções. Para tanto, falaremos sobre suas nomenclaturas e funcionamento assim como os seus retornos de valores e parâmetros.

2.1 Conceitos e características de uma função

Funções tem o objetivo de encapsular instruções de forma a possibilitar a modularização do código. Inicialmente, podemos definir modularização como a decomposição funcional de um sistema computacional. Porque falamos "sistema computacional" e não "programa"? Um sistema computacional pode envolver vários programas intercambiando informações. A figura a seguir ilustra uma abstração de um sistema modular.

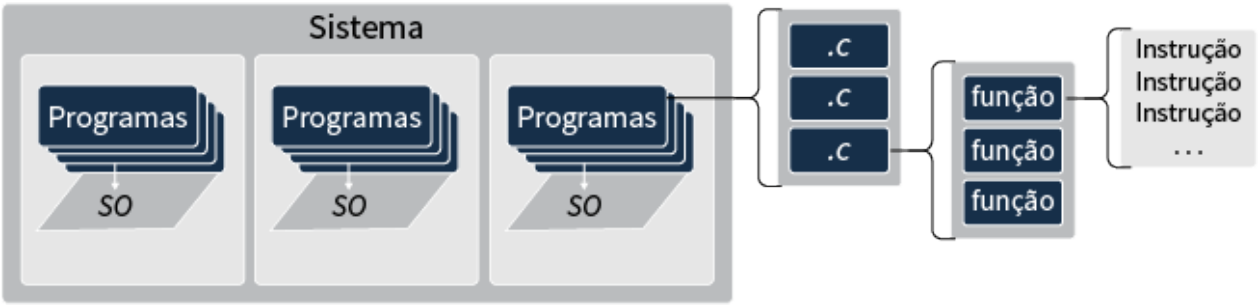

Figura 1 - Sistema modular formado por vários programas que podem estar em execução em dispositivos distintos – cada dispositivo suportado pelo seu respectivo sistema operacional.
Fonte: Elaborada pelo autor, 2019.

Na figura acima, o sistema é formado por vários programas que utilizam as funcionalidades exportadas pelo sistema operacional em execução no respectivo dispositivo computacional. Tais funcionalidades são acessíveis por intermédio das funções do sistema, disponibilizadas através das bibliotecas padrões. Cada programa, por sua vez, pode ser composto por vários arquivos-fonte (arquivos "*.c*"). Em função da complexidade do programa, é sugerível a sua decomposição em vários arquivos de codificação, separados por funcionalidades. Por exemplo, tais funcionalidades poderão ser distinguidas entre aquelas que objetivam a interação com o usuário, aquelas que realizam acesso à um banco de dados e aquelas que fazem os processamentos internos do sistema. Essas funcionalidades, por sua vez, são conseguidas através da codificação de funções específicas.

VOCÊ O CONHECE?

A ideia de modularização surgiu no final da década de 1960, na chamada "crise do software". Na ocasião, Dijkstra já defendia a ideia de modularização tendo inclusive, apresentado em 1972, na ACM (*Association for Computing Machinery* – literalmente, em português, Associação para Máquinas de Computação), o trabalho intitulado "*The Humble Programmer*" (O Humilde Programador). Para saber um pouco mais sobre Dijkstra, você poderá acessar este artigo aqui < http://horizontes.sbc.org.br/index.php/2017/10/23/807/>. (MOCHETTI, 2017).

Essa ideia de dividir o programa em funções é um dos conceitos da modularização de sistemas, mais especificamente, modularização de programas.

VOCÊ QUER VER?

A modularização dos programas ajuda a torná-lo mais fácil de escrever, de testar e de reaproveitar códigos, podendo torná-lo, também, mais eficiente. O vídeo a seguir < https://www.infoq.com/br/presentations/modularizacao-de-codigo-c/> aborda o tema de modularização em linguagem C. (LAVRATTI, 2014)

Mas, quais são as vantagens de uma programação modular, contendo inclusive funções em sua codificação? Modularizar não significa apenas deixar o código organizado, mas sim, apresenta uma série de vantagens, segundo Deitel, (2011). Para conhecer quais são essas vantagens, clique nas abas abaixo.

- **Implementação**

  Facilita o processo de criação, testes e correção de erros: a implementação, testes e correções, com o uso de módulos fica mais pontual, ou seja, pode-se focar apenas em uma funcionalidade específica; desta forma, consegue-se por exemplo, corrigir erros mais facilmente pois a localização do código, dentre milhares de linhas de código, fica mais rápida. Com funções é possível trocar toda uma função por outra de versão acima.

- **Incorporação**

  Possibilidade de reutilização do código: o bloco representado pela função poderá ser utilizado em outro projeto bastando incorporá-la ao código.

- **Função**

  Evitar reescrita do código ao longo do arquivo-fonte: caso uma certa ação representada por diversas linhas de código seja requerida em vários momentos e em vários pontos do programa, não será necessário reescrever o código e sim, apenas chamar a função.

Até então falamos de alguns conceitos inerentes às funções mas, na prática, será que já usamos funções nesta disciplina além da função "**main()**"? A resposta é positiva, por exemplo, o "**printf**" é uma função da chamada biblioteca padrão; que evoca funcionalidades do sistema operacional para intervir nas ações que culmina na manipulação do sistema de vídeo do dispositivo computacional. Outros exemplos de funções que já utilizamos são: **scanf**, **strlen**, **strcmp**, **strcpy**, **atoi** e **itoa**.

VOCÊ SABIA?

Você sabia que as funções poderão ser evocadas remotamente? Em sistemas ditos como sistemas distribuídos, um programa pode evocar uma função remota, localizada em outro computador. Essa chamada é realizada por meio de *RPC* (*Remote Procedure Call*, ou em português, Chamada a Procedimentos Remotos). Para ler sobre o assunto e, inclusive, ver códigos-exemplo em C, você pode acessar este *link* <http://www.eletrica.ufpr.br/pedroso/2009/TE090/Aulas/rpc.pdf>. Em Java, existe algo semelhante chamado *RMI* (*Remote Method Invocation*, em português, Invocação de Método Remoto). (PEDROSO, 2006).

Mas, como criar funções? A seguir veremos mais detalhes de como criar as funções e como elas funcionam no escopo de sua execução no dispositivo computacional.

2.2 Nome de uma função

Da mesma forma das variáveis, uma função deve ter um nome para que seja evocado. As regras para a definição do nome são as mesmas em relação às variáveis; clique nos itens abaixo e confira.

Não pode conter caracteres especiais (como por exemplo: "$", "@", e acentos).
Deve começar por uma letra ou com um traço "underline" ("_").
Para facilitar a implementação, o seu nome deve ser sugestivo, ou seja, ser compatível com a sua funcionalidade.

Para a definição de uma função, deve-se seguir a seguinte sintaxe:

```
<tipo_de_retorno> nome_da_função (parâmetros)
```

Para exemplificar a sintaxe, tomemos como exemplos:

```
void ImprimeMensagens(char mens[])
int Fatorial(int N)
int main ( )
```

Mas, o que é tipo de retorno e o que são parâmetros de uma função? Porque o “main” tem um tipo de retorno “int”? E os parâmetros do main? Uma função pode ter ausência de parâmetros? Antes de conversarmos sobre essas dúvidas, vamos falar sobre como funcionam as funções.

2.3 Como Funciona uma Função

Quando uma função é evocada, deve haver um desvio de fluxo de processamento de modo a executar as linhas de código da função chamada. A figura a seguir ilustra o processo de execução de uma função.

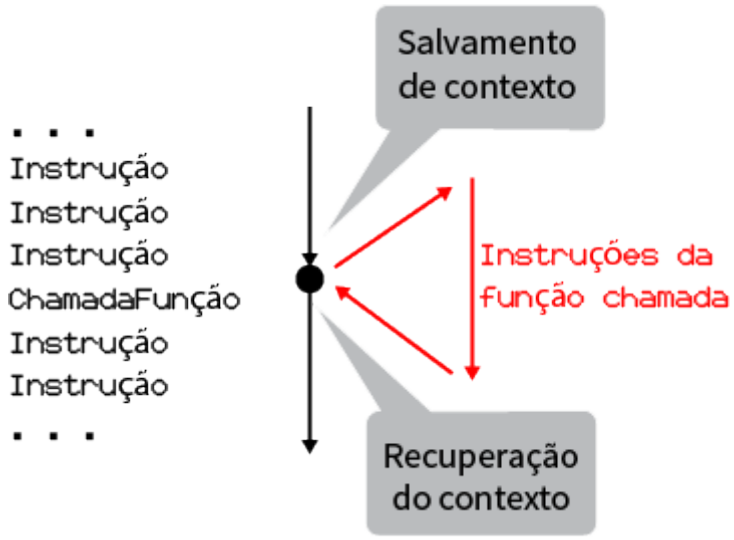

Figura 2 - Cenário criado para a chamada de uma função. O fluxo é desviado para que as instruções pertinentes à função sejam executadas.
Fonte: Elaborada pelo autor, 2019.

Na figura acima, são destacados os eventos de salvamento e recuperação de contexto; estes eventos são responsáveis por salvar e recuperar em uma região da memória principal mapeada na forma de pilha de dados, o valor dentre outras informações, do registrador PC (*program conter*, ou, contador de programa). O registrador PC, também chamado de IP (*instruction pointer*, ou, ponteiro de instrução) é uma estrutura de *hardware* que armazena o endereço da linha a ser executada. Na chamada da função, esse valor corresponde ao ponto que houve a evocação da função: esse ponto será restaurado para que a execução do programa continue na linha imediatamente após a chamada da função.
Uma outra questão para entendermos sobre o funcionamento das funções diz respeito às variáveis; declarar uma variável dentro da função é a mesma coisa que declararmos fora? Para respondermos à tal questionamento, veremos, a seguir, o corpo de uma função.

2.4 O Corpo de uma função

Um ponto que merece destaque na implementação de uma função reside no fato de que as variáveis declaradas dentro da função são denominadas como variáveis locais. Variáveis locais somente são visíveis dentro da própria função enquanto que as variáveis globais são visíveis em toda extensão do programa. Para entendermos melhor essa diferença, tomemos por base o trecho de código a seguir:
int x;

```
. . .
int funcao1( )
{
int a,b;
a = x;
b = y;
}
int main( )
{
int y;
. . .
}
```

No trecho acima, encontramos a declaração de "**x**" fora de qualquer função, enquanto a declaração de "**y**" encontra-se dentro do corpo da "**main()**". Podemos ver, ainda, uma função "**funcao1**", que manipula tanto "**x**" quanto "**y**" para atribuir os seus valores às variáveis "**a**" e "**b**", respectivamente. Como a variável "**x**" é global, a "**funcao1**" poderá acessá-la em seu código, porém, haverá um erro de compilação na linha "**b = y**" do tipo "*variável **y** não declarada*" pois, pelo fato de ser local a "**main**", não é visível à "**funcao1**".

Uma variável local existe na memória apenas durante a execução da função. Quando a função acaba o seu processamento e o fluxo voltam para o ponto de chamada, as variáveis locais são apagadas da memória. Caso a função seja chamada localmente, serão alocadas, novamente, as variáveis locais sendo os seus valores iniciados novamente; os valores da execução antiga são perdidos, a menos que se use o modificador "**static**" (por exemplo, "**static int a**").

Para ilustrar melhor uma função, vamos supor o código a seguir:

```
#include <stdio.h>
int x;
void funcaoTeste( )
{
int a;
a = x;
printf("Valor de 'a': %d\n",a);
x++;
}
int main( )
{
x = 0;
funcaoTeste();
printf("Valor de 'x': %d\n",x);
}
```

No exemplo acima, temos a função "**funcaoTeste()**" que cria a variável local "**a**" e manipula, também, a variável global "**x**". A função "**funcaoTeste()**" é evocada a partir da função "**main()**" – após a sua finalização, ocorre a execução da linha que contém o código: "**printf("Valor de 'x': %d\n",x);**".

Você deve estar se perguntando: Mas, e o tipo da função? Para que serve, por exemplo, o "**int**" antes do "**main()**"? Veremos agora a questão de retorno das funções.

2.5 Funções que não retornam valor

Inicialmente, quando pensamos em uma função matemática, nos vem à cabeça um valor retornado, por exemplo, a função matemática "*seno*" retorna o valor do seno de um ângulo. Dessa forma, podemos já fazer uma analogia

entre a função matemática e a função computacional: o tipo associado à função refere-se ao formato do valor retornado pela função. Voltando ao caso do *seno*, o seu retorno seria o tipo "**float**".
Mas, toda função deve retornar um valor? O retorno de valor de uma função não é um item obrigatório; neste caso, uma função consiste apenas de uma sequência de ações sem o retorno do resultado de seu processamento.
Para alguns autores, como Ascencio (2012), em outras linguagens como a linguagem PASCAL, diferencia-se uma função com retorno e uma função sem retorno. Para se adequar à linguagem, uma função que nada retorna é chamada de procedimento ("*procedure*"). Já de acordo com Puga (2016), a diferença básica entre procedimento e função consiste no fato de que as funções poderão ser usadas em expressões, ou seja, usadas como parte geradora de um valor que será atribuído à uma variável. No caso do procedimento, como ele não retorna valor, não será possível utilizá-lo em expressões de atribuição.

VOCÊ QUER LER?

A função "printf", que você já vem utilizando em seus programas, é passível de várias formatações de saída, não se limitando ao tipo de dado a ser manipulado. Para saber um pouco mais a respeito do comando "**printf**", veja o material disponível aqui <https://www.vivaolinux.com.br/artigo/Parametros-interessantes-do-scanf-e-do-printf-em-C>. (SCHLEMER, 2009)

No caso da linguagem *C/C++*, a diferenciação se faz apenas na definição do tipo do retorno. O uso é definido pelo programador, por exemplo, a função "**printf**" geralmente é usada fora de expressões, porém, ela retorna a quantidade de caracteres impressos e poderia ser usada da seguinte forma:

```
qtd_carac_impressos = printf("Olá mundo!");
```

Como podemos perceber pela linha de código acima, a linguagem *C/C++* não faz muita distinção entre uma função retornando ou não retornando valor. Mas, como diferenciar na implementação? Para exemplificar, vamos dar uma olhada no código colocado abaixo:

```c
#include <stdio.h>

void Espera()
 {
   unsigned int i=0,j;
   for(; i<50000; i++)
     for(j=0;j<50000; j++);
 }

int main()
 {
   printf("Vamos esperar um pouco...\n");
   Espera();
   printf("E esperar mais um pouco...\n");
   Espera();
   printf("Testando sua paciencia... a espera derradeira...\n");
   Espera();
   printf("Pronto! Voce eh agora uma pessoa mais paciente! :-)\n");
   return 0;
 }
```

No código acima, definimos como "**void**" o tipo do retorno da função "**Espera( )**". Quando se define uma função com o tipo "**void**" (vazio) não necessitamos usar o "**return <valor>**" ao final de seu processamento. Caso seja necessário interromper o processamento de uma função do tipo "**void**", basta inserir um "**return**" sem valor de retorno, ou seja, simplesmente: "**return;**".
Vejamos mais um exemplo:

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>  //para usar o toupper()


char mensagem[30], opcao='0';

void EntradaMenu()
{
  do
    {
      printf("\nDigite 'X' para Processar X\n");
      printf("Digite 'Y' para processar Y\n");
      printf("Digite 'E' para encerrar.\n");
      scanf("%d",&opcao);
      opção = toupper(opcao);  //toupper() transforma o parâmetro em maiúsculo.
    }while ((opcao!='X')&&(opcao!='Y')&&(opcao!='E'));
}

void ImprimirMensagem()
{
  printf("Mensagem: %s\n",mensagem);
}

int main()
{
  EntradaMenu();
  switch(opcao)
    {
      case 'X':
                strcpy(mensagem,"Processamento de X");
                break;
      case 'Y':
                strcpy(mensagem,"Processamento de Y");
                break;
      case 'E':
                strcpy(mensagem,"Encerrando programa.");
                break;
    }
  ImprimirMensagem();
  return 0;
}
```

No exemplo acima, as funções "**EntradaMenu( )**" e "**ImprimirMensagem( )**" não possuem retorno e as informações são intercambiadas por intermédio de variáveis globais (variáveis definidas fora do bloco representado pelo "**main()**". Chamamos esse tipo de implementação de acoplamento forte, pois existem as funções que dependem da existência das variáveis globais. Uma consequência deste tipo de dependência é a dificuldade de manutenção, pois qualquer alteração, por exemplo, nas variáveis globais, afetam todas as funções delas dependentes.

Todas as funções que não possuem retorno dependem de variáveis globais para o seu funcionamento? Não, a dependência ou não de variáveis globais dependerá exclusivamente de suas funcionalidades para as quais uma função é implementada. Mas, existem funções que retornam valores? Sim, veremos como implementar funções que retornam valores a seguir.

2.6 Funções que retornam valor

Voltando à nossa abstração matemática, a função "*seno*" é uma função que retorna valor. Em C, no caso de visualizarmos na função uma linha com o comando "**return <valor>**", como acontece em "**main()**" ("**int main()... return 0**"), podemos dizer que a referida função retorna valor. Mas então, qual seria o retorno do "**main()**"? Quem chamou o "**main()**" para receber o valor de retorno? Lembre-se que o "**main()**" é o ponto de partida de um programa escrito em C/C++, portanto, ele é chamado pelo sistema operacional. O valor inteiro retornado pelo "**main()**" ao sistema operacional, representa um valor indicativo do motivo de sua finalização: normal ou decorrente de uma falha de execução. Esse código de retorno permite ao sistema operacional realizar ações de controle e, dependendo do caso, ativar ações de interação com o usuário. Mas, como retornar um valor? Creio que você esteja lembrado da linha do "**main()**" com o código "**return 0**". O retorno de valor é realizado através do "**return**". Para relembrar o "**return**" no "**main()**", segue um trecho:

```
int main( )
{
int y;
. . .
return 0;
}
```

Pode-se aplicar a mesma estrutura visto na função "main()" para outras funções? Sim, o programador pode criar funções conforme a sua necessidade, o que lhe dá, também, a possibilidade de especificar o tipo de retorno conforme a sua conveniência. Por exemplo, vamos dar uma olhada no código para gerar seis dezenas para apostarmos na Mega-sena:

```
#include <stdio.h> //para a funcao printf()
#include <stdlib.h> //para as funcoes srand() e rand()
#include <time.h> //para a funcao time()
int DezenaSorteada()
{
int dezena;
do
{
dezena = rand()%61; //o valor 60 é válido
} while(dezena==0); //evitar retornar o valor 0
return dezena;
}
int main( )
{
int i=0,qtd;
srand(time(NULL));
printf("Quantidade de dezenas a serem sorteadas: ");
scanf("%d",&qtd);
for (; i < qtd ; i++)
{
printf("Sorteio [%d]: %d\n",i+1,DezenaSorteada());
```

```
}
return 0;
}
```

No código acima temos um exemplo de uma função que retorna um valor inteiro. No caso específico do exemplo, a função "**DezenaSorteada()**" retorna um valor entre 1 e 60 para que seja impresso na tela para o usuário por intermédio da linha "**printf("Sorteio [%d]: %d\n",i+1,DezenaSorteada());**". Para tanto, foram utilizadas as funções "**srand**", que inicia o gerador de números aleatórios, "**time()**" que retornará o tempo transcorrido desde 1 de janeiro de 1970 (em segundos) – que servirá como "semente" para a iniciação do gerador de números aleatórios e, por fim, a função "**rand()**", que retornará um valor inteiro entre 0 e a constante "**RAND_MAX**" (definida no arquivo *header* "*stdlib.h*").

O código com "**return**" deve ser inserido apenas na última linha de uma função? Não necessariamente. Pode-se colocar "**return**" ao longo do corpo da função, para interromper a função e voltar ao ponto de chamada caso algum erro tenha ocorrido. Um exemplo de "**return**" ao longo do corpo da função está referenciado no trecho de código a seguir:

```
int funcao( )
{
int x,y;
. . .
if(y == 0)
return -1;
. . .
return abs(x / y);
}
```

No exemplo acima, temos o término antecipado da função em virtude da situação de erro testada no comando condicional (no caso, o denominador da divisão não pode assumir o valor 0). Ao ser verificada essa condição de erro, a função retorna o valor "-1". Foi escolhido esse valor como retorno de erro pois, em operação normal, a função sempre retornará um valor positivo representado pelo valor absoluto da divisão de "**x**" por "**y**". O valor absoluto é conseguido através da utilização da função "**abs()**", definida no arquivo *header* "*stdlib.h*". Essa finalização feita antes de se chegar ao final da função poderá ser usada, também, em casos normais de execução, onde um certo objetivo já tenha sido cumprido. Você pode ficar a se perguntar: eu já ouvi falar em um comando chamado "**break**" (usado nos laços de repetição e, inclusive no comando "**switch...case**"). O "**break**" tem a mesma funcionalidade em relação ao "**return**"? Para falarmos sobre essa dúvida, vamos analisar o código abaixo:

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

int Busca()
 {
   char str[9]="abcdefgh", valor_buscado;
   int i;
   printf("Valor a ser procurado (ou '0' para finalizar o programa): ");
   scanf("%c",&valor_buscado);
   if(valor_buscado=='0')
      return -1;
   if((valor_buscado<'a')||(valor_buscado>'z'))
      return -2;
   for(i=0; i<strlen(str); i++)
      if(str[i]==valor_buscado)
         break;
   if(i!=strlen(str))
      return i;
   return -3;
 }

int main()
 {
   int retorno;
   do
     {
       retorno = Busca();
       switch(retorno)
     {
       case -2:
                  printf("Valor digitado fora da faixa.\n");
                              break;
       case -1:
                  break;
       case -3:
                  printf("Valor nao encontrado.\n");
                  break;
       default:
                  printf("Encontrado na posicao %d\n",retorno);
     }
     }while(retorno!=FINAL_PROCESSAMENTO);
   return 0;
 }
```

No código acima, temos o uso de "**return**" e de "**break**". Podemos ver que, ao utilizar-se o "**return**", não é colocada a parte "**else**" nos comandos condicionais pelo fato de que a execução é interrompida, retornando à

posição de chamada da função (no caso, retornando ao "**main**"). Por sua vez, acontece a interrupção do laço, continuando na própria função quando usado o "**break**". Desta forma, a função retornará um valor positivo indicando a posição do elemento buscado dentro do vetor ou retornará um valor negativo indicando uma operação malsucedida. Para finalizar, foram sublinhadas no código, todas as ocorrências de funções para que possamos identificar e diferenciar o que é comando e o que é função dentro da linguagem *C/C++*. Vamos então reescrever o exemplo do menu de opções (mostrado quando estávamos falando sobre funções que não retornam valor) usando funções que retornam valor?

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

char mensagem[30];

char EntradaMenu()
{
  char opcao='0';
  do
    {
      printf("\nDigite 'X1' para Processar X\n");
      printf("Digite 'Y2' para processar Y\n");
      printf("Digite 'E3' para encerrar.\n");
      scanf("%d",&opcao);
      opção = toupper(opcao);
    }while ((opcao!='X')&&(opcao!='Y')&&(opcao!='E'));
  return opcao;
}

. . .

int main()
{
  char op;
  op=EntradaMenu();
  . . .
```

Na versão do código acima, podemos notar que a variável "**opcao**" desapareceu da lista de variáveis globais. O intercâmbio referente à escolha do usuário passou a ser por intermédio do retorno da função "**EntradaMenu( )**". Sendo assim, a coesão, em relação a essa função passou a ser baixa pois não há dependências de variáveis globais. No caso do exemplo acima, a função retorna um tipo "**opcao**"; desta forma, podemos mencionar que uma função pode retornar qualquer tipo de dados previamente definido. Vamos ver um outro exemplo? Para tanto, suponha o código abaixo, de uma calculadora, para realizar as quatro operações aritméticas básicas:

```c
#include <stdio.h>

int a, b;  //numeros a serem manipulados

int soma()
{
  return a+b;
}

int sub()
{
  return a-b;
}

int mul()
{
  return a*b;
}

int div()
{
  return a/b;
}

char Menu()
{
  char opcao;
  do
   {
     printf("Digite:\n\t'+' para soma\n\t'-' para subtracao\n");
     printf("\t'*' para multiplicacao\n\t'/' para divisao:\n");
     printf("Opcao:  ");
     scanf("%c",&opcao);
   } while((opcao!='+')&&(opcao!='-')&&(opcao!='*')&&(opcao!='/'));
  return opcao;
}

int main()
{
  char op;
  int res;
  printf("Valor de 'a':  ");
  scanf("%d",&a);
  printf("Valor de 'b':  ");
  scanf("%d",&b);
```

```
    op=Menu();
    switch(op)
      {
        case '+':
                res=soma();
                break;
        case  -':
                  res=sub();
                break;
        case '*':
                  res=mul();
                break;
        case '/':
                  res=div();
                break;
      }
    printf("Resultado:  %d\n", res);
    return 0;
}
```

Neste código, cada operação aritmética é representada por uma função. Para simplificar o exemplo, não estão sendo tratadas condições de erro, como por exemplo, a divisão por 0. A partir da escolha realizada pelo usuário, feita por intermédio da função "**Menu()**", a instanciação da variável "**res**" é feita com o retorno da função correspondente.

Até o momento, falamos apenas sobre o retorno das funções. Mas, como passar valores para as funções de modo que elas possam coletar e processar valores diferentes a cada chamada? Isso é o veremos a seguir, quando conversaremos sobre "parâmetros" das funções.

2.7 Parâmetros

Como já antecipamos um pouco, parâmetros servem para informar à função qual deve ser o conjunto de valores que precisam ser processados. Para ficar mais claro, vamos voltar ao exemplo da função matemática "*seno*". Ao usarmos "*y=seno(x)*" já fica subtendido que desejamos atribuir à variável "*y*" o valor do seno referente a "*x*" graus. Desta forma, podemos falar que "*x*" é o parâmetro da função "*seno*". Na computação não é diferente, ou seja, parâmetro são os valores passados para as funções para que elas possam realizar os seus processamentos específicos.

Antes de entrarmos especificamente nos parâmetros das funções, vamos nos prender um pouco na função "**main()**". Será que podemos passar argumentos quando evocamos o programa por intermédio da linha de comando (prompt ou cmd)? Sim, a função "**main()**" consegue receber parâmetros, basta implementar como sugere o trecho de código a seguir:

```
int main(int argc, char *argv[])
 {
   if(argc==1)
     {
       printf("Parametros não passados...");
       return 0;
     }
   printf("Nome do programa chamado: %s\n",argv[0]);
   printf("Parametro passado: %s\n",argv[1]);
   . . .
```

A função "**main()**" admite dois parâmetros passados pela linha de comando: o primeiro parâmetro, do tipo inteiro, indica a quantidade de argumentos passados. Caso o usuário não passe parâmetros, a variável (no nosso exemplo, chamada como "argc") tem o valor 1. O valor 1 refere-se ao próprio nome do programa passado como parâmetro pelo sistema operacional – que ocupa a posição 0 do vetor, "**argv**", que contém a lista de parâmetros. Os demais parâmetros, quando existirem, poderão ser acessados nas posições subsequentes à posição 0.
Mas, e o caso das funções que não sejam a função "**main()**"? Como devemos passar informações para elas? Para a passagem de parâmetros em uma função deveremos criar uma lista de variáveis em sua interface na seguinte forma:
<tipo_de_retorno> NomeDaFunção (lista de parâmetros)
Sendo que a lista de parâmetros se assemelha à criação de variáveis, ou seja, as variáveis deverão ser criadas de acordo com a sintaxe:
(<tipo1> var1, <tipo2> var2, . . .)
Por exemplo, no caso termos uma função que receba, como parâmetros, dois valores inteiros para seja realizada e retornado o valor de uma soma, teríamos:

```
int Soma(int a, int b)
```

A ordem de definição das variáveis deverá ser a mesma em relação à chamada da função, ou seja, o primeiro parâmetro passado será associado ao primeiro parâmetro da interface da função, o segundo da chamada ao segundo da interface e assim por diante. Cada parâmetro poderá ser passado por valor ou por referência, conforme será descrito a seguir. (MIZRAHI, 2008).

2.8 Parâmetros passados por valor e por referência

Como mencionado anteriormente, existem duas formas de se passar um parâmetro: por valor e por referência. Para adiantar, vamos falar, genericamente, que a passagem por valor é usada quando o parâmetro é apenas de entrada, ou seja, permite que a informação seja passada apenas para dentro da função. A passagem por referência autoriza que o fluxo seja bidirecional, ou seja, aceita que a informação seja passada para a função e a partir dela. Daremos sequência ao aprofundamento do assunto a seguir.

2.8.1 Parâmetros passados por valor

A forma mais simples de se passar parâmetros para a função é através de passagem por valor. Neste tipo de passagem, o sistema realiza uma cópia do valor passado em uma variável criada localmente. Para continuarmos a detalhar sobre o assunto, tomemos como exemplo o código adaptado de Puga (2016):

```
#include <stdio.h>

int soma_dobro(int a, int b)
 {
   a = a * 2;
   b = b * 2;
   return a+b;
 }

int main()
 {
   int x,y,res;
   printf("Digite o valor de X: ");
   scanf("%d",&x);
   printf("Digite o valor de Y: ");
   scanf("%d",&y);
   res=soma_dobro(x,y);
   printf("Resultado do processamento de %d e %d:  %d", x , y, res);
   return 0;
 }
```

No código acima, temos uma função ("**soma_dobro**") que recebe dois parâmetros: "**a**" e "**b**". Como sabemos que os parâmetros são passados por valor? No caso, podemos verificar que cada parâmetro é definido pela seguinte sintaxe:

```
<tipo_retorno_da_função>  nome_função (<tipo> nome_do_parâmetro)
```

Mas, alterando-se dentro da função os valores das variáveis passadas como parâmetros ("**a**" e "**b**"), não serão alterados os valores das variáveis na origem da chamada (no caso, as variáveis "**x**" e "**y**")? Não há esse perigo, pois, quando se passar por valor, cada parâmetro origina uma variável local que é instanciada com o valor passado pela variável na origem da chamada. Neste caso, por exemplo, as variáveis "**x**" e "**a**" são totalmente distintas. A figura a seguir mostra melhor o que acabamos de falar.

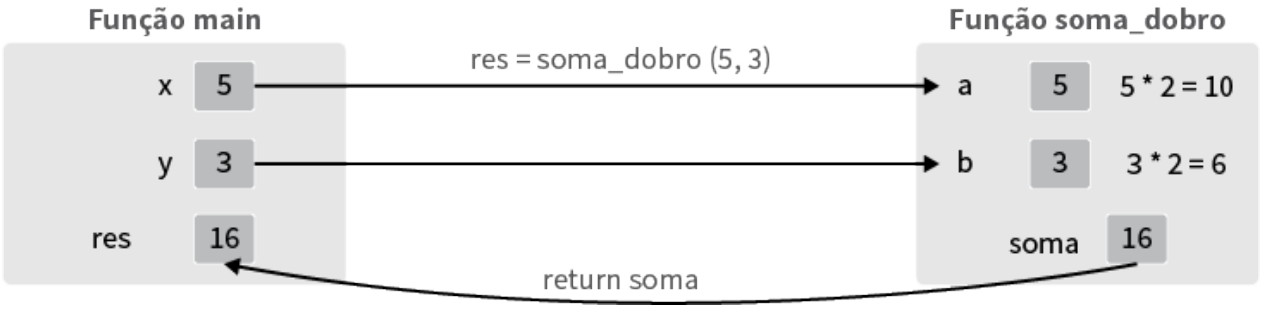

Figura 3 - Passagem de parâmetros por valor. Cada parâmetro da função implica na criação de uma variável local que recebe uma cópia do valor passado como parâmetro, não afetando a variável que originou a informação.
Fonte: PUGA, 2016. p. 267.

Para finalizar, podemos falar que utilizamos passagem por valor quando desejamos manter as informações intactas no ponto de chamada ou, ainda, quando a função não objetiva alterações nos valores dos parâmetros mas sim, apenas usá-los como fonte de dados para prosseguir o seu processamento.

2.8.2 Parâmetros passados por referência

Como mencionamos anteriormente, os valores dos parâmetros passados por valor são mantidos intactos quando estamos nos referenciando ao ponto de chamada da função. Mas, e se desejarmos, por algum motivo, alterar os valores das variáveis no ponto de origem? Para isso, temos a opção de realizar passagem por referência. Vamos modificar o código utilizado para a passagem de parâmetros por valor e transformá-lo para passagem de parâmetros por referência:

```
#include <stdio.h>

int soma_dobro(int *a, int *b)
 {
   *a = *a * 2;
   *b = *b * 2;
   return *a+*b;
 }

int main()
 {
   int x,y,res;
   printf("Digite o valor de X: ");
   scanf("%d",&x);
   printf("Digite o valor de Y: ");
   scanf("%d",&y);
   res=soma_dobro(&x,&y);
   printf("Resultado do processamento de %d e %d:  %d", x , y, res);
   return 0;
 }
```

Para começar, notamos que aparece o símbolo "*" na definição dos parâmetros da função. O símbolo "*" ((int *a, int *b)) indica que estamos, ao invés de um valor, recebendo uma posição de memória cujo conteúdo será manipulado pela função. Esse mesmo sinal aparece quando manipulamos as variáveis no corpo da função. Em tal ocasião, o sinal "*" denota o conteúdo da posição de memória. Por exemplo, na linha "*a = *a * 2;" ;" podemos traduzir como: o conteúdo da posição de memória apontada por "**a**" recebe o conteúdo da posição de memória apontado por "**a**" vezes 2. Desta forma, como estamos atribuindo um valor para uma posição de memória, automaticamente estamos alterando o valor da variável que originou o parâmetro na chamada da função (no caso do exemplo, a variável "**x**").

Já que estamos falando que "***x**" indica uma posição de memória, significa que temos que passar para a função não um valor e, sim, uma posição de memória. Este feito é conseguido utilizando-se o sinal "&". No caso do exemplo, a chamada "soma_dobro(&x,&y) " pode ser traduzida como: evoca-se a função "**soma_dobro**" passando como parâmetros a posição de memória apontada por "**x**" e a posição de memória apontada por "**y**". Creio que agora, você está começando a entender sobre o motivo de usarmos o símbolo "**&**" na utilização da função "**scanf**". No caso, passamos o endereço da variável passada pela "**scanf**" para que a função possa nos devolver o valor fornecido pelo usuário.

A figura a seguir ilustra esse processo de passagem por parâmetro tomando por base o exemplo de código acima.

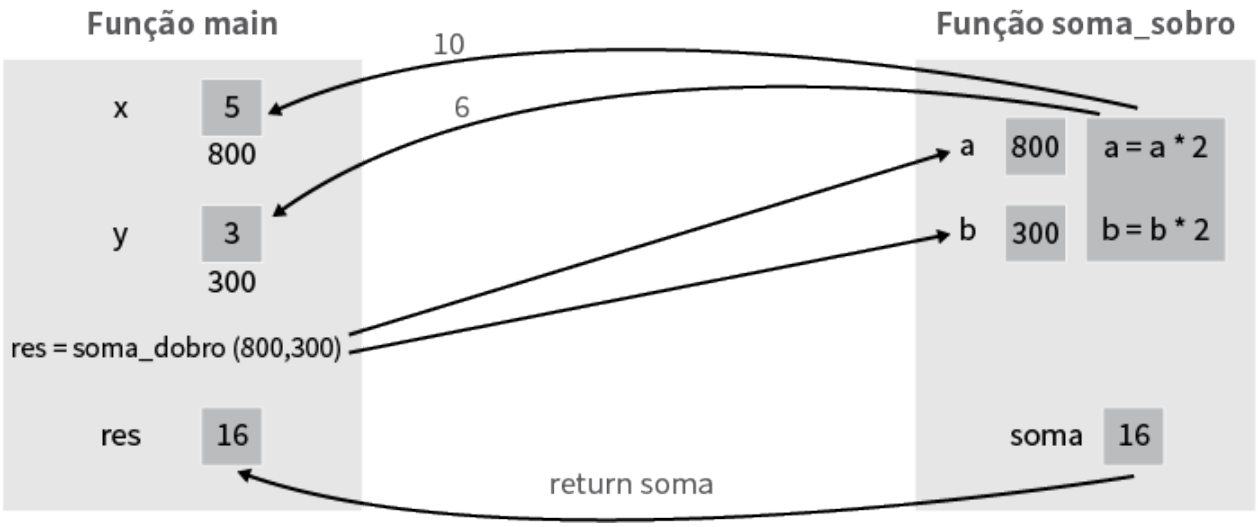

Figura 4 - Passagem de parâmetros por referência. Cada parâmetro representa uma posição de memória cujo endereço é o mesmo da variável passada na chamada da função, desta forma, qualquer alteração sobre os parâmetros afeta as variáveis que originaram a informação.
Fonte: PUGA, 2016. p. 268.

No caso da figura acima, está sendo suposto que as variáveis "**x**" e "**y**" estão localizadas nas posições 800 e 300 da memória, respectivamente. Sendo assim, qualquer alteração dentro da função "**soma_dobro**" será realizada exatamente em tais posições de memória, modificando, consequentemente, os valores de "**x**" e de "**y**".

2.8.3 Arquivo de cabeçalho (arquivo header)

Já mencionamos, em diversas ocasiões, sobre os arquivos de cabeçalho "*.h*" (arquivos *header*). Mas, qual a diferença entre um arquivo *header* e um arquivo de código? Porque ele aparece em grande parte dos programas? Os arquivos de cabeçalho servem para que as definições (por exemplo, de constantes, estruturas de dados e protótipos de funções), a serem utilizadas pelo programa sejam feitas. Como um arquivo "*.c*" pode chamar, por exemplo, funções implementadas em outro arquivo "*.c*", o compilador deve conhecer a estrutura da função

chamada, ou seja, saber o tipo de retorno e os tipos dos parâmetros, para verificar durante o processo de compilação, se existe algum tipo de inconsistência em relação ao uso da função.
Para montar um arquivo ".h" tomemos por exemplo, o código anterior, no qual colocaremos o cabeçalho da função "soma dobro". Neste caso, o arquivo ficaria assim:

```
#ifndef SOMADOBRO
#define SOMADOBRO "somadobro.h"

int soma_dobro(int *, int *);

#endif
```

O "**#ifndef**" serve para testar se já foi definido a constante "**SOMADOBRO**"; caso já tenha sido definida, aborta-se a inclusão do arquivo "*.h*" específico, senão, define-se a constante "**SOMADOBRO**" por intermédio do "**#define**" (pode-se colocar qualquer valor associado a esta constante), e inclui o protótipo da função nas regras de compilação. O "**#ifndef**" evita que ocorra duplicidade de definições quando o arquivo "*.h*" for referenciado por vários arquivos "*.c*".
Em relação ao código-fonte, a única diferença é que apareceria a linha:

```
#include "somadobro.h"
```

Utilizar passagem de parâmetros nas funções impacta o fator de seu acoplamento. Acoplamento relaciona-se ao grau de interdependência dos módulos, ou seja, como os módulos compartilham, por exemplo, variáveis globais; no caso, alterações nas estruturas de tais variáveis impacta diretamente sobre os módulos tendo que, neste caso, haver modificações em todos os módulos que as utiliza. Utilizar passagem de parâmetros nas funções representa a construção de módulos com acoplamento fraco. Acoplamento fraco significa uma maior independência entre os módulos e, consequentemente, às variáveis compartilhadas globalmente.

CASO

Um profissional da área de computação, ao receber um projeto, ficou a pensar em como organizar e estruturar o seu código. Ao verificar o alto grau de complexidade do sistema, começou a refletir sobre a modularização. Imediatamente, já lhe veio à cabeça os conceitos de acoplamento e de coesão: acoplamento no sentido de como os módulos/funções iriam intercambiar informações e, coesão, no sentido de não haver, por exemplo, sobreposição das funcionalidades. Diante disso, mesmo sem definir o estilo de programação (estruturada ou orientada a objetos), a primeira medida que tomou foi aplicar conceitos da Engenharia de Software. Para tanto, ele estudou padrões e modelos como: GRASP (*General Responsibility Assignment Software Principles* – Princípios Gerais de Atribuição e de Responsabilidade do *Software*), UML (*Unified Modeling Language* – Linguagem de Modelagem Unificada) e o princípio SRP do SOLID (***S**ingle Responsibility Principle* – Princípio da Responsabilidade Única). Em suma, ele resolveu fazer esse exercício de abstração pois, um sistema deve ser otimizado, bem definido e bem estruturado, independentemente de qual paradigma de linguagem será adotado.

Construir uma função com fraco acoplamento torna o reaproveitamento do código e alterações em seu código mais fáceis. Tal facilidade é conseguida pois toda evocação é baseada em passagem de parâmetros, ou seja, sem a necessidade de dependência, por exemplo, de variáveis globais.

Síntese

Chegamos ao fim do nosso segundo encontro sobre técnicas de programação. Tivemos agora, a oportunidade de ampliar os conceitos e funcionalidades da programação estruturada, mais especificamente, da programação usando a linguagem C. Com os pontos abordados, você já conseguirá implementar programas mais complexos e torná-los mais eficientes e estruturados pela utilização de técnicas de modularização utilizando funções. Com os temas estudados até aqui, esperamos que você continue treinando e incrementando os seus programas computacionais de forma a deixá-los mais eficientes e organizados.
Nesta unidade, você teve a oportunidade de:

- ter contato com conceitos de modularização de programas;
- definir funções corretamente analisando tipo de retorno e parâmetros a serem passados;
- decidir as ocasiões que poderão ser alvo da modularização.

Bibliografia

ASCENCIO, A. F. G. **Fundamentos de Programação de Computadores**: Algoritmos, PASCAL, C/C++ (Padrão ANSI) e Java. 3. Ed. São Paulo: Pearson Education do Brasil, 2012. Disponível em: <https://laureatebrasil.blackboard.com/>. Acesso em: 08/07/2019.

DEITEL, P. J.; DEITEL, H. **C: Como Programar**. 6. ed. São Paulo: Pearson Prentice Hall, 2011. Disponível em: <https://laureatebrasil.blackboard.com/>. Acesso em: 08/07/2019.

LAVRATTI, F. **Modularização de Código C**. 30 min. São Paulo: TDC. The Developers Conference, 2014. Disponível em: <https://www.infoq.com/br/presentations/modularizacao-de-codigo-c/>. Acesso em: 08/07/2019.

MIZRAHI, V. V. **Treinamento em Linguagem C**. 2. ed. São Paulo: Pearson Prentice Hall, 2008. Disponível em <https://laureatebrasil.blackboard.com/>. Acesso em 08/07/2019.

MOCHETTI, K. Dijkstra, um Pioneiro em todas as Áreas. **Revista digital SBC Horizontes**. 2017. Disponível em: <http://horizontes.sbc.org.br/index.php/2017/10/23/807/>. Acesso em 08/07/2019.

PEDROSO, C. M. **Laboratório de Redes. Remote Procedure Call – RPC**. 2006. Disponível em: <http://www.eletrica.ufpr.br/pedroso/2009/TE090/Aulas/rpc.pdf>. Acesso em: 08/07/2019.

PUGA, S.; RISSETTI, G. **Lógica de Programação e Estruturas de Dados – com Aplicações em Java**. 3 ed. São Paulo: Pearson Education do Brasil, 2016. Disponível em: <https://laureatebrasil.blackboard.com/>. Acesso em: 08/07/2019.

SCHLEMER, E. **Parâmetros Interessantes do scanf e do printf em C**. 2009. Disponível em: <https://www.vivaolinux.com.br/artigo/Parametros-interessantes-do-scanf-e-do-printf-em-C>. Acesso em: 08/07/2019.