Funções

Alexandre Mello

Fatec Campinas

2018

Roteiro

Funções

- Um ponto chave na resolução de um problema complexo é conseguir "quebrá-lo"em subproblemas menores.
- Ao criarmos um programa para resolver um problema, é crítico quebrar um código grande em partes menores, fáceis de serem entendidas e administradas.
- Isto é conhecido como modularizacão, e é empregado em qualquer projeto de engenharia envolvendo a construção de um sistema complexo.

Funções

Funções

São estruturas que agrupam um conjunto de comandos, que são executados quando a função é chamada/invocada.

- Vocês já usaram algumas funções como **scanf** e **printf**.
- Algumas funções podem devolver algum valor ao final de sua execução:

```
x = sqrt(4);
```

- Vamos aprender como criar e usar funções.

Porque utilizar funções?

- Evitar que os blocos do programa fiquem grandes demais e, por conseqüência, mais difíceis de ler e entender.
- Separar o programa em partes que possam ser logicamente compreendidas de forma isolada.
- Permitir o reaproveitamento de código já construído (por você ou por outros programadores).
- Evitar que um trecho de código seja repetido várias vezes dentro de um mesmo programa, minimizando erros e facilitando alterações.

Definindo uma função

Uma função é definida da seguinte forma:

```
tipo_retorno nome(tipo parâmetro1,..., tipo parâmetroN){
    comandos;
    return valor_de_retorno;
}
```

- Toda função deve ter um tipo (**int, char, float, void, etc**). Esse tipo determina qual será o tipo de seu valor de retorno.
- Os **parâmetros** são variáveis, que são inicializadas com valores indicados durante a invocação da função.
- O comando **return** devolve para o invocador da função o resultado da execução desta.

Definindo uma função: Exemplo 1

A função abaixo recebe como parâmetro dois valores inteiros. A função faz a soma destes valores, e devolve o resultado.

```
int soma (int a, int b) {
   int c;
   c = a + b;
   return c;
}
```

- Note que o valor de retorno (variável **c**) é do mesmo tipo da função.
- Quando o comando **return** é executado, a função para de executar e retorna o valor indicado para quem fez a invocação (ou chamada) da função.

Definindo uma função: Exemplo 1

```
int soma (int a, int b) {
   int c;
   c = a + b;
   return c;
}
```

- Qualquer função pode invocar esta função, passando como parâmetro dois valores inteiros, que serão atribuídos para as variáveis **a** e **b** respectivamente.

```
int main(){
  int r;
  r = soma(12, 90);
  r = soma (-9, 45);
}
```

Definindo uma função: Exemplo 1

```
#include <stdio.h>

int soma (int a, int b) {
  int c;
  c = a + b;
  return c;
}


int main(){
  int r;
  r = soma(12, 90);
  printf("r = %d\n", r);
  r = soma (-9, 45);
  printf("r = %d\n", r);
}
```

Definindo uma função: Exemplo 2

- A lista de parâmetros de uma função pode ser vazia.

```
int leNumero() {
  int c;
  printf("Digite um número:");
  scanf("%d", &c);
  return c;
}
```

- O retorno será usado pelo invocador da função:

```
int main(){
  int r;
  r = leNumero();
  printf("Numero digitado: %d\n", r);
}
```

Definindo uma função: Exemplo 2

```
#include <stdio.h>

int leNumero() {
  int c;
  printf("Digite um numero:");
  scanf("%d", &c);
  return c;
}

int main(){
  int r;
  r = leNumero();
  printf("Numero digitado: %d\n", r);
}
```

Exemplo de função 3

```
#include <stdio.h>
int soma(int a, int b){
  int c;
  c = a + b;
  return c;
}

int main(){
  int res, x1=4, x2=-10;
  res = soma(5,6);
  printf("Primeira soma: %d\n",res);
  res = soma(x1,x2);
  printf("Segunda soma: %d\n",res);
}
```

- Qualquer programa começa executando os comandos da função **main**.
- Quando se encontra a chamada para uma função, o fluxo de execução passa para ela e se executa os comandos até que um **return** seja encontrado ou o fim da função seja alcançado.
- Depois disso o fluxo de execução volta para o ponto onde a chamada da função ocorreu.

Exemplo de função 4

- A expressão contida dentro do comando **return** é chamado de valor de retorno (é a resposta da função). Nada após ele será executado.

```
#include <stdio.h>

int leNumero() {
  int c;
  printf("Digite um numero:");
  scanf("%d", &c);
  return c;
  printf("Bla bla bla!\n");
}

int soma (int a, int b) {
  int c;
  c = a + b;
  return c;
}

int main(){
  int x1, x2, res;
  x1 = leNumero();
  x2 = leNumero();
  res = soma(x1, x2);
  printf("Soma e: %d\n", res);
}
```

- Não será impresso *Bla bla bla!*

Invocando uma função

- Uma forma clássica de realizarmos a invocação (ou chamada) de uma função é atribuindo o seu valor à uma variável:

```
x = soma(4, 2);
```

- Na verdade, o resultado da chamada de uma função é uma expressão e pode ser usada em qualquer lugar que aceite uma expressão:

Exemplo

```
printf("Soma de a e b: %d\n", soma(a, b));
```

Invocando uma função

- Na chamada da função, para cada um dos parâmetros desta, devemos fornecer um valor de mesmo tipo, e na mesma ordem dos parâmetros.

```
#include <stdio.h>

int somaComMensagem(int a, int b, char st[100]){
  int c = a+b;
  printf("%s = %d\n", st, c);
  return c;
}

int main(){
  somaComMensagem(4, 5, "Resultado da soma:");
}
```

- A saída do programa será:

```
Resultado da soma: = 9
```

- Já a chamada abaixo gerará um erro de compilação.

```
int main(){
  somaComMensagem(4, "Resultado da soma:",  5);
}
```

Invocando uma função

- Ao chamar uma função passando variáveis **simples** como parâmetros, estamos usando apenas os seus valores que serão copiados para as variáveis parâmetros da função.
- Os valores das variáveis na chamada da função não são afetados por alterações dentro da função.

```
#include <stdio.h>

int incr(int x){
  x = x + 1;
  return x;
}

int main(){
  int a = 2, b;
  b = incr(a);
  printf("a = %d, b = %d\n", a, b);
}
```

- O que será impresso? O valor de **a** é alterado pela função **incr**?

Invocando uma função

- Veremos passagem de vetores como parâmetros posteriormente, mas é bom ressaltar que variáveis do tipo vetores podem ser alteradas quando passadas como parâmetro para uma função!

```
#include <stdio.h>

void printVet(int v[5]){
  int i;
  for(i=0; i<5; i++){
    printf("%d, ", v[i]);
  }
  printf("\n");
  v[0] = 9;
}

int main(){
  int vet[]={1, 2, 3, 4, 5};
  printVet(vet);
  printVet(vet);
}
```

- O programa irá imprimir:

```
1, 2, 3, 4, 5,
9, 2, 3, 4, 5,
```

O tipo **void**

- O tipo **void** é um tipo especial.
- Ele representa "nada", ou seja, uma variável desse tipo armazena conteúdo indeterminado, e uma função desse tipo retorna um conteúdo indeterminado.
- Em geral este tipo é utilizado para indicar que uma função não retorna nenhum valor.

O tipo **void**

- Por exemplo, a função abaixo imprime o número que for passado para ela como parâmetro e não devolve nada.
- Neste caso não utilizamos o comando **return**.

```
void imprime (int numero){
  printf ("Número %d\n", numero);
}
```

O tipo **void**

```
#include <stdio.h>

void imprime(int numero){
  printf ("Número %d\n", numero);
}

int main (){
  imprime(10);
  imprime(20);
  return 0;
}
```

A função **main**

- O programa principal é uma função especial, que possui um tipo fixo (**int**) e é invocada automaticamente pelo sistema operacional quando este inicia a execução do programa.
- Quando utilizado, o comando **return** informa ao sistema operacional se o programa funcionou corretamente ou não. O padrão é que um programa retorne zero caso tenha funcionado corretamente ou qualquer outro valor caso contrário.

Exemplo

```
int main() {
  printf("Hello, World!\n");
  return 0;
}
```

Protótipo de funções: definindo funções depois do **main**

- Até o momento, aprendemos que devemos definir as funções antes do programa principal. O que ocorreria se declarássemos depois?

```
#include <stdio.h>

int main () {
  float a = 0, b = 5;
  printf ("%f\n", soma (a, b));
  return 0;
}

float soma (float op1, float op2) {
  return (op1 + op2);
}
```

- Dependendo do compilador, ocorre um erro de compilação!

Protótipo de funções: declarando uma função sem defini-la

- Para organizar melhor um programa, e podermos implementar funções em partes distintas do arquivo fonte, utilizamos **protótipos de funções**.
- Protótipos de funções correspondem a primeira linha da definição de uma função contendo: tipo de retorno, nome da função, parâmetros e por fim **um ponto e vírgula**.

```
tipo_retorno nome(tipo parâmetro1,..., tipo parâmetroN);
```

- O protótipo de uma função deve aparecer antes do seu uso.
- Em geral coloca-se os protótipos de funções no início do seu arquivo do programa.

Em geral o programa é organizado da seguinte forma:

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

Protótipos de funções

int main(){
  Comandos;
}

int fun1(Parâmetros){
  Comandos;
}

int fun2(Parâmetros){
  Comandos;
}
...
...
```

Protótipo de Funções: Exemplo 1

```
#include <stdio.h>

float soma(float op1, float op2);
float subt(float op1, float op2);

int main () {
  float a = 0, b = 5;
  printf (" soma = %f\n subtracao = %f\n", soma (a, b), subt(a, b));
  return 0;
}

float soma (float op1, float op2) {
  return (op1 + op2);
}

float subt (float op1, float op2) {
  return (op1 - op2);
}
```

Funções Podem Invocar Funções

- Nos exemplos anteriores apenas a função **main** invocava funções por nós definidas.
- Isto não é uma regra. Qualquer função pode invocar outra função (exceto a **main** que é invocada apenas pelo sistema operacional).
- Veja o exemplo no próximo slide.

Funções Podem Invocar Funções

- Note que **fun1** invoca **fun2**, e isto é perfeitamente legal.
- O que será impresso?

```
#include <stdio.h>

int fun1(int a);
int fun2(int b);

int main(){
  int c = 5;
  c = fun1(c);
  printf("c = %d\n", c);
}

int fun1(int a){
  a = a + 1;
  a = fun2(a);
  return a;
}

int fun2(int b){
  b = 2*b;
  return b;
}
```

Exercício

- Escreva uma função que computa a potência a^b para valores a (double) e b (int) passados por parâmetro (não use bibliotecas como math.h). Sua função deve ter o seguinte protótipo:

double pot(double a, int b);

- Use a função anterior e crie um programa que imprima todas as potências:

$$2^0, 2^1, \ldots, 2^{10}, 3^0, \ldots, 3^{10}, \ldots, 10^{10}.$$

Exercício

- Escreva uma função que computa o fatorial de um número n passado por parâmetro. Sua função deve ter o seguinte protótipo:

  long fat(long n); OBS: Caso $n \leq 0$ seu programa deve retornar 1.
- Use a função anterior e crie um programa que imprima os valores de $n!$ para $n = 1, \ldots, 20$.

Variáveis locais e variáveis globais

- Uma variável é chamada local se ela foi declarada dentro de uma função. Nesse caso ela existe somente dentro da função, e após o término da execução desta, a variável deixa de existir. **Variáveis parâmetros também são variáveis locais**
- Uma variável é chamada global se ela for declarada fora de qualquer função. Essa variável é visível em todas as funções. Qualquer função pode alterá-la e ela existe durante toda a execução do programa.

Organização de um Programa

- Em geral um programa é organizado da seguinte forma:

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

Protótipos de funções

Declaração de Variáveis Globais

int main(){
  Declaração de variáveis locais
  Comandos;
}

int fun1(Parâmetros){ //Parâmetros também são variáveis locais
  Declaração de variáveis locais
  Comandos;
}

int fun2(Parâmetros){ //Parâmetros também são variáveis locais
  Declaração de variáveis locais
  Comandos;
}
...
...
```

Escopo de variáveis

- O escopo de uma variável determina de quais partes do código ela pode ser acessada, ou seja, de quais partes do código a variável é visível.
- A regra de escopo em C é bem simples:
  - As variáveis globais são visíveis por todas as funções.
  - As variáveis locais são visíveis apenas na função onde foram declaradas.

Escopo de variáveis

```
#include<stdio.h>

void fun1();
int fun2(int local_b);

int global;

int main() {
  int local_main;
  /* Neste ponto são visíveis global e local_main */
}

void fun1() {
  int local_a;
  /* Neste ponto são visíveis global e local_a */
}

int fun2(int local_b){
  int local_c;
  /*Neste ponto são visíveis global, local_b e local_c*/
}
```

Escopo de variáveis

- É possível declarar variáveis locais com o mesmo nome de variáveis globais.
- Nesta situação, a variável local "esconde" a variável global.

```
#include <stdio.h>

void fun();

int nota = 10;

int main(){
   nota = 20;
   fun();
}

void fun() {
  int nota;
  nota = 5;
  /* Neste ponto nota é a variável local de fun. */
}
```

Exemplo 1

```
#include <stdio.h>

void fun1();
void fun2();

int x;
int main(){
  x = 1;
  fun1();
  fun2();
  printf("main: %d\n", x);
}

void fun1(){
  x = x + 1;
  printf("fun1: %d\n",x);
}

void fun2(){
  int x = 3;
  printf("fun2: %d\n",x);
}
```

O que será impresso ?

Exemplo 2

```
#include <stdio.h>

void fun1();
void fun2();

int x = 1;
int main(){
  int x=1;
  fun1();
  fun2();
  printf("main: %d\n", x);
}

void fun1(){
  x = x + 1;
  printf("fun1: %d\n",x);
}

void fun2(){
  int x = 4;
  printf("fun2: %d\n",x);
}
```

O que será impresso ?

Exemplo 3

```
#include <stdio.h>

void fun1();
void fun2(int x);

int x = 1;
int main(){
  x=2;
  fun1();
  fun2(x);
  printf("main: %d\n", x);
}

void fun1(){
  x = x + 1;
  printf("fun1: %d\n",x);
}

void fun2(int x){
  x = x + 1 ;
  printf("fun2: %d\n",x);
}
```

O que será impresso ?

Variáveis locais e variáveis globais

- O uso de variáveis globais deve ser evitado pois é uma causa comum de erros:
  - Partes distintas e funções distintas podem alterar a variável global, causando uma grande interdependência entre estas partes distintas de código.
- A legibilidade do seu código também piora com o uso de variáveis globais:
  - Ao ler uma função que usa uma variável global é difícil inferir seu valor inicial e portanto qual o resultado da função sobre a variável global.

Exemplo Utilizando Funções

- Em uma das aulas anteriores vimos como testar se um número em **candidato** é primo:

```
divisor = 2;
eprimo=1;
while(divisor<=candidato/2) {
  if(candidato % divisor == 0){
    eprimo=0;
    break;
  }
  divisor++;
}
if(eprimo)
  printf(" %d, ", candidato);
```

Exemplo Utilizando Funções

- Depois usamos este código para imprimir os *n* primeiros números primos:
- Veja no próximo slide.

Exemplo Utilizando Funções

```
int main(){
  int divisor=0, n=0, eprimo=0, candidato=0, primosImpr=0;
  printf("\n Digite numero de primos a imprimir:");
  scanf("%d",&n);
  if(n>=1){
    printf("2, ");
    primosImpr=1;
    candidato=3;
    while(primosImpr < n){
      divisor = 2;
      eprimo=1;
      while( divisor <= candidato/2 ){
        if(candidato % divisor == 0){
          eprimo=0;
          break;
        }
        divisor++;
      }
      if(eprimo){
        printf("%d, ",candidato);
        primosImpr++;
      }
      candidato=candidato+2;//Testa proximo numero
    }
  }
```

Exemplo Utilizando Funções

- Podemos criar uma função que testa se um número é primo ou não (note que isto é exatamente um bloco logicamente bem definido).
- Depois fazemos chamadas para esta função.

Exemplo Utilizando Funções

```
int ePrimo(int candidato){
  int divisor;

  divisor = 2;
  while( divisor <= candidato/2){
    if(candidato % divisor == 0){
      return 0;
    }
    divisor++;
  }
  //Se terminou o laço então candidato é primo
  return 1;
}
```

Exemplo Utilizando Funções

```
#include <stdio.h>

int ePrimo(int candidato); //retorna 1 se candidato é primo, e 0 caso contrário

int main(){
  int n=0, candidato=0, primosImpr=0;

  printf("Digite numero de primos:");
  scanf("%d",&n);
  if(n >= 1){
     printf("2, ");
     primosImpr = 1;
     candidato = 3;
     while(primosImpr < n){
       if( ePrimo(candidato) ){
         printf("%d, ",candidato);
         primosImpr++;
       }
       candidato=candidato+2;
     }
  }
}
```

Vetores em funções

- Vetores também podem ser passados como parâmetros em funções.
- Ao contrário dos tipos simples, vetores têm um comportamento diferente quando usados como parâmetros de funções.
- Quando uma variável simples é passada como parâmetro, seu valor é atribuído para uma nova variável local da função.
- No caso de vetores, **não é criado** um novo vetor!
- Isto significa que os valores de um vetor **são alterados** dentro de uma função!

Vetores em funções

```
#include <stdio.h>

void fun1(int vet[], int tam){
  int i;
  for(i=0;i<tam;i++)
     vet[i]=5;
}

int main(){
  int x[10];
  int i;

  for(i=0;i<10;i++)
    x[i]=8;

  fun1(x,10);
  for(i=0;i<10;i++)
    printf("%d\n",x[i]);
}
```

O que será impresso?

Vetores em funções

- No exemplo anterior note que a função **fun1** recebe o vetor como parâmetro e um inteiro que especifica o seu tamanho.

```
void fun1(int vet[], int tam){
  int i;
  for(i=0;i<tam;i++)
     vet[i]=5;
}
```

- Esta é a forma padrão para se receber um vetor como parâmetro.
- Um vetor possui um tamanho definido, mas em geral usa-se menos posições do que o seu tamanho. Além disso a função pode operar sobre vetores de diferentes tamanhos, bastando informar o tamanho específico de cada vetor na variável **tam**.

Vetores em funções

- Vetores não podem ser devolvidos por funções.

```
#include <stdio.h>

int[] leVet() {
  int i, vet[100];
  for (i = 0; i < 100; i++) {
    printf("Digite um numero:");
    scanf("%d", &vet[i]);
  }
  return vet;
}
```

- O código acima não compila, pois não podemos retornar um **int[]** .

Vetores em funções

- Mas como um vetor é alterado dentro de uma função, podemos criar a seguinte função para leitura de vetores.

```
#include <stdio.h>

void leVet(int vet[], int tam){
  int i;
  for(i = 0; i < tam; i++){
    printf("Digite numero:");
    scanf("%d",&vet[i]);
  }
}
```

- A função abaixo faz a impressão de um vetor.

```
void escreveVet(int vet[], int tam){
  int i;
  for(i=0; i< tam; i++)
    printf("vet[%d] = %d\n",i,vet[i]);
}
```

Vetores em funções

- Podemos usar as funções anteriores no programa abaixo.

```
int main(){
  int vet1[10], vet2[20];

  printf(" ------ Lendo Vetor 1 --------\n");
  leVet(vet1,10);
  printf(" ------ Lendo Vetor 2 --------\n");
  leVet(vet2,20);

  printf(" ------ Imprimindo Vetor 1 --------\n");
  escreveVet(vet1,10);
  printf(" ------ Imprimindo Vetor 2 --------\n");
  escreveVet(vet2,20);

}
```

Vetores multi-dimensionais e funções

- Ao passar um **vetor simples** como parâmetro, não é necessário fornecer o seu tamanho na declaração da função.
- Quando o **vetor é multi-dimensional** a possibilidade de não informar o tamanho na declaração se restringe à primeira dimensão apenas.

```
void mostra_matriz(int mat[][10], int n) {
   ...
}
```

Vetores multi-dimensionais e funções

- Pode-se criar uma função deixando de indicar a primeira dimensão:

```
void mostra_matriz(int mat[][10], int n) {
   ...
}
```

- Ou pode-se criar uma função indicando todas as dimensões:

```
void mostra_matriz(int mat[10][10], int n) {
   ...
}
```

- Mas não pode-se deixar de indicar outras dimensões (exceto a primeira):

```
void mostra_matriz(int mat[10][], int n) {
  //ESTE NÃO FUNCIONA
   ...
}
```

Vetores multi-dimensionais e funções

- É comum definirmos uma constante com o tamanho máximo de matrizes e vetores multi-dimensionais, e passarmos os tamanhos efetivamente utilizados como parâmetros para funções que operam sobre matrizes ou vetores-multidimensionais.

```
#include <stdio.h>
#define MAX 10

void imprimeMatriz(int mat[MAX][MAX], int lin, int col) {
  int i, j;

  for (i = 0; i < lin; i++) {
    for (j = 0; j < col; j++)
      printf("%d\t", mat[i][j]);
    printf("\n");
  }
}
```

Vetores multi-dimensionais em funções

```
#include <stdio.h>
#define MAX 10

void imprimeMatriz(int mat[MAX][MAX], int lin, int col) {
  int i, j;

  for (i = 0; i < lin; i++) {
    for (j = 0; j < col; j++)
      printf("%d\t", mat[i][j]);
    printf("\n");
  }
}

int main() {
  int mat[MAX][MAX] = { { 0,  1,  2,  3,  4,  5},
                        {10, 11, 12, 13, 14, 15},
                        {20, 21, 22, 23, 24, 25},
                        {30, 31, 32, 33, 34, 35},
                        {40, 41, 42, 43, 44, 45},
                        {50, 51, 52, 53, 54, 55},
                        {60, 61, 62, 63, 64, 65},
                        {70, 71, 72, 73, 74, 75}};

  imprimeMatriz(mat, 8, 6);
  return 0;
}
```

Vetores multi-dimensionais em funções

- Lembre-se que vetores (multi-dimensionais ou não) são alterados quando passados como parâmetro em uma função.

```
void teste(int mat[MAX][MAX], int lin, int col) {
  int i, j;

  for (i = 0; i < lin; i++) {
    for (j = 0; j < col; j++){
         mat[i][j] = -1;
    }
  }
}

int main() {
  int mat[MAX][MAX] = { { 0, 1},
                        { 2, 3} };
  teste(mat, 2, 2);
  return 0;
}
```

- Qual o conteúdo de **mat** após a execução da função **teste**?

Exercício

- Escreva uma função em C para computar a raiz quadrada de um número positivo. Use a idéia abaixo, baseada no método de aproximações sucessivas de Newton. A função deverá retornar o valor da vigésima aproximação.

Seja Y um número, sua raiz quadrada é raiz da equação

$$f(x) = x^2 - Y.$$

A primeira aproximação é $x_1 = Y/2$. A $(n+1)$-ésima aproximação é

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Exercício

- Escreva uma função em C que recebe como parâmetros duas matrizes quadradas $n \times n$ e computa a soma destas ($n \leq 100$). O protótipo da função deve ser:

```
void somaMat(double mat1[100][100], double mat2[100][100],
                                    double matRes[100][100], int n);
```

- As matrizes **mat1** e **mat2** devem ser somadas e o resultado atribuído à **matRes**. O parâmetro **n** indica as dimensões das matrizes.

Exercício

- Escreva uma função em C que recebe como parâmetros duas matrizes quadradas $n \times n$ e computa a multiplicação destas ($n \leq 100$). O protótipo da função deve ser:

```
void multiplicaMat(double mat1[100][100], double mat2[100][100],
                                          double matRes[100][100], int n);
```

- As matrizes **mat1** e **mat2** devem ser multiplicadas e o resultado atribuído à **matRes**. O parâmetro **n** indica as dimensões das matrizes.