

MC102 – Algoritmos e Programação de Computadores

Instituto de Computação

UNICAMP

Primeiro Semestre de 2014

Roteiro

- 1 Funções
- 2 O tipo void
- 3 A função `main`
- 4 Protótipo de funções
- 5 Exemplo de uso de funções
- 6 Escopo de variáveis
- 7 Vetores e funções
- 8 Matrizes e funções
- 9 Exercícios

Funções

- Um aspecto importante na resolução de um problema complexo é conseguir dividi-lo em subproblemas menores.
- Ao criarmos um programa para resolver um determinado problema, uma tarefa crítica é dividir o código grande em partes menores, fáceis de serem compreendidas e mantidas.

Funções

Funções

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

Exemplo:

```
scanf ("%d", &x);
```

Funções

As funções podem retornar um valor ao final de sua execução.

Exemplo:

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

Por que utilizar funções?

- Evitar que os blocos do programa fiquem grandes demais e, por consequência, 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, evitando inconsistências e facilitando alterações.

Definindo uma função

Uma função é definida da seguinte forma:

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

- Toda função deve ter um tipo que determina seu valor de retorno.
- Os parâmetros são variáveis que serão utilizadas pela função. Tais variáveis são inicializadas com valores na chamada de execução da função.

Exemplo de função

A função abaixo soma dois valores, passados como parâmetros:

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

- O valor de retorno deve ser do mesmo tipo definido para a função.
- Quando o comando `return` é executado, a função termina sua execução e retorna o valor indicado para quem fez a chamada da função.

Exemplo de função

```
#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);

    printf("Segunda soma: %d\n", soma(x1, x2));

    return 0;
}
```

Importante: a execução de um programa sempre começa pela função main.

Definindo uma função

Uma função pode não ter parâmetros, neste caso, basta não informá-los:

```
#include <stdio.h>

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

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

int main() {
    int x1, x2;
    x1 = leNumero();
    x2 = leNumero();
    printf("Valor da soma: %d\n", soma(x1, x2));

    return 0;
}
```

Definindo uma função

A expressão contida no comando `return` é chamada de valor de retorno. Nenhum comando após o `return` será executado.

```
#include <stdio.h>

int leNumero() {
    int n;
    printf("Digite um numero: ");
    scanf("%d", &n);
    return n;
    printf("bla bla bla bla!"); /* Nao imprime esta mensagem */
}

int main() {
    int x1, x2;
    x1 = leNumero();
    x2 = leNumero();
    printf("Soma: %d\n", x1 + x2);
    return 0;
}
```

Definindo uma função

- Funções só podem ser definidas fora de outras funções.
- Lembre-se que o programa principal (`main`) é uma função.

Invocando uma função

Uma forma comum de realizarmos a invocação (ou chamada) de uma função é atribuindo o seu valor a 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

- Para cada um dos parâmetros da função, devemos fornecer um valor, de mesmo tipo, na chamada da função.
- Ao chamar uma função passando variáveis 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.

Invocando uma função

```
#include <stdio.h>

int somaEsquisita(int x, int y) {
    x = x + 1;
    y = y + 1;
    return (x + y);
}

int main() {
    int a = 10, b = 5;

    printf("Soma de a e b: %d\n", a + b);
    printf("Soma de x e y: %d\n", somaEsquisita(a, b));
    printf("a: %d\n", a);
    printf("b: %d\n", b);

    return 0;
}
```

Os valores de a e b não são alterados por operações feitas em x e y !

O tipo void

- O tipo `void` é um tipo especial.
- Este tipo é utilizado para indicar que uma função não retorna nenhum valor.

O tipo void

- Por exemplo, a função abaixo imprime o número inteiro que for passado para ela como parâmetro:

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


O tipo void

```
#include <stdio.h>

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

int main() {
    imprime(10);
    imprime(20);

    return 0;
}
```

O tipo void

- Podemos usar o comando `return` (sem qualquer valor) para indicar explicitamente o fim de uma função do tipo `void`.

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

O tipo void

```
#include <stdio.h>

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

    return;
}

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;  
}
```

Definindo funções depois do `main`

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

Declarando funções depois do main

```
#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);
}
```

Ocorrerá um erro de compilação, já que a função soma não foi definida antes do ponto em que ela foi invocada.

Declarando uma função sem defini-la

- Para organizar melhor um programa e podermos implementar funções em partes distintas do arquivo, *protótipos de funções* são utilizados.
- Protótipos de funções correspondem à primeira linha da definição de uma função contendo tipo de retorno, nome da função, parâmetros e um ponto e vírgula.

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

- O protótipo de uma função deve vir sempre antes do seu uso.
- É comum colocar os protótipos de funções no início do arquivo do programa.

Protótipos de funções

```
#include <stdio.h>

float soma(float op1, float op2);

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

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


Protótipos de funções

```
#include <stdio.h>

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

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

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

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

Números primos

- Em aulas anteriores, vimos como testar se um número é primo:

```
divisor = 2;  
primo = 1;
```

```
while (primo && (divisor <= candidato / 2)) {  
    if (candidato % divisor == 0)  
        primo = 0;  
    divisor++;  
}
```

```
if (primo)  
    printf("%d", candidato);
```

Números primos

- Vimos também como escrever um programa para imprimir os n primeiros números primos (veja o próximo slide).

Números primos

```
#include <stdio.h>

int main() {
    int divisor = 0, n = 0, primo = 0, candidato = 2, encontrados = 0;

    printf("Numero de primos a calcular: ");
    scanf("%d", &n);

    while (encontrados < n) {
        divisor = 2;
        primo = 1;
        while (primo && (divisor <= candidato / 2)) {
            if (candidato % divisor == 0) primo = 0;
            divisor++;
        }
        if (primo) {
            printf("%d\n", candidato);
            encontrados++;
        }
        candidato++;
    }
    return 0;
}
```

Números primos

- Se o número de primos a ser impresso é negativo, usaremos o valor absoluto deste.
- Como refazer este código 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).
- Vamos criar também uma função que retorna o valor absoluto de um número.
- Depois fazemos chamadas para estas funções.

Números primos

```
#include <stdio.h>

int testaPrimo(int candidato);
int valorAbs(int x);

int main() {
    int divisor = 0, n = 0, primo = 0, candidato = 2, encontrados = 0;

    printf("Numero de primos a imprimir: ");
    scanf("%d", &n);
    n = valorAbs(n);

    while (encontrados < n) {
        if (testaPrimo(candidato)) {
            printf("%d\n", candidato);
            encontrados++;
        }
        candidato++;
    }

    return 0;
}
```

Números primos

```
/* Calcula o valor absoluto de x */  
int valorAbs(int x) {  
    if (x < 0)  
        return -x;  
    else  
        return x;  
}
```

Números primos

```
/* Verifica se um numero candidato eh primo */
int testaPrimo(int candidato) {
    int divisor, primo;
    divisor = 2;
    primo = 1;

    while (primo && (divisor <= candidato / 2)) {
        if (candidato % divisor == 0)
            primo = 0;
        divisor++;
    }

    if (primo)
        return 1; /* Se for primo, retorna 1 (verdadeiro) */
    else
        return 0; /* Se nao for, retorna 0 (falso) */
}
```


Números primos

- As funções permitem maior clareza dos códigos.
- Também é simples de fazer modificações.
- Exemplo: melhorar o teste de primalidade.
 - ▶ Testar se o candidato é um número par maior que 2 (não é primo).
 - ▶ Se for ímpar, testar apenas divisores ímpares (3, 5, 7, etc).
- O uso de funções facilita a manutenção do código. Neste caso, altera-se apenas a função `testaPrimo`.

Números primos

```
int testaPrimo(int candidato) {
    int divisor, primo;

    if (candidato % 2 == 0)
        return (candidato == 2);

    divisor = 3;
    primo = 1;
    while (primo && (divisor <= candidato / 2)) {
        if (candidato % divisor == 0)
            primo = 0;
        divisor = divisor + 2;
    }

    return primo;
}
```

Variáveis locais e variáveis globais

- Uma variável é chamada local se ela foi declarada dentro de uma função. Nesse caso, ela existe somente dentro daquela função e, após o término da execução da mesma, a variável deixa de existir. Variáveis utilizadas como parâmetros de funções também são 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.

Variáveis locais e variáveis globais

```
#Declaração de bibliotecas
```

```
#Definição de constantes
```

```
Protótipos de funções;
```

```
Declaração de variáveis globais;
```

```
int main() {
```

```
    Declaração de variáveis locais;
```

```
    Comandos;
```

```
}
```

```
int funcao1(parâmetros) { /* Parâmetros também são considerados locais */
```

```
    Declaração de variáveis locais;
```

```
    Comandos;
```

```
}
```

```
int funcao2(parâmetros) { /* Parâmetros também são considerados locais */
```

```
    Declaração de variáveis locais;
```

```
    Comandos;
```

```
}
```

Escopo de variáveis

- O escopo de uma variável determina em quais partes do código pode-se ter acesso a ela.
- 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 funcao1();
int funcao2(int local_b);

int global_a;

int main() {
    int local_main;
    /* Neste ponto sao visiveis global_a e local_main */
}

void funcao1() {
    int local_a;
    /* Neste ponto sao visiveis global_a e local_a */
}

int funcao2(int local_b) {
    int local_c;
    /* Neste ponto sao visiveis global_a, 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.

```
int nota = 10;

void a() {
    int nota;

    ...

    /* Altera o valor da variavel local,
       sem afetar a variavel global */
    nota = 5;
}
```

Escopo de variáveis

```
#include <stdio.h>
int x = 1;

void funcao1() {
    x = 3;
    printf("%d ", x);
}

void funcao2() {
    int x = 4;
    printf("%d ", x);
}

int main() {
    x = 2;
    funcao1();
    funcao2();
    printf("%d\n", x);
    return 0;
}
```

O que será impresso por este programa?

3 4 3

Vetores e 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 podem ser alterados dentro de uma função.

Vetores e funções

- Para indicar que um parâmetro é um vetor, usamos [] na frente do nome do parâmetro. Exemplo:

```
int max(int vetor[], int tam) {  
    ...  
}
```

- Ao chamar uma função que possui um vetor como parâmetro, devemos apenas indicar o nome do vetor a ser fornecido para a função (sem usar [] na frente do nome do vetor). Exemplo:

```
int elementos[10], n;  
...  
x = max(elementos, n);
```

Vetores e funções

```
#include <stdio.h>

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

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

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

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

    return 0;
}
```

O que será impresso por este programa?

O programa imprimirá o valor 5 dez vezes, cada valor em uma linha.

Vetores e funções

- Vetores não podem ser usados como retorno de uma função.

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

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

- Entretanto, podemos fazer algo semelhante usando o fato de que vetores são alterados dentro de funções.

Vetores e funções

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

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

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

Vetores e funções

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

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

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

    return 0;
}
```

Matrizes e funções

- Ao passar um vetor como parâmetro, não é necessário fornecer o seu tamanho na declaração da função.
- Quando usamos uma matriz, a possibilidade de não informar o tamanho na declaração se restringe à primeira dimensão.
- Como vimos anteriormente, matrizes são de fato alocadas pelo compilador como vetores linearizados, por isso é preciso informar todas as dimensões (com exceção da primeira) para que o compilador seja capaz de determinar corretamente os elementos de uma matriz passada como parâmetro para uma função.
- Assim como vetores, matrizes passadas como parâmetros podem ser modificadas dentro das funções.

Matrizes e funções

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

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

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

```
void mostra_matriz(int mat[5][10], int linhas) {  
    ...  
}
```

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

```
void mostra_matriz(int mat[5][], int linhas) {  
    /* Este programa nao compila */  
    ...  
}
```


Exemplo

```
#include <stdio.h>

void imprime_matriz(int linhas, int colunas, int matriz[][10]) {
    int i, j;
    for (i = 0; i < linhas; i++) {
        for (j = 0; j < colunas; j++)
            printf("%2d ", matriz[i][j]);
        printf("\n");
    }
}

int main() {
    int matriz[8][10] = { { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                          {10, 11, 12, 13, 14, 15, 16, 17, 18, 19},
                          {20, 21, 22, 23, 24, 25, 26, 27, 28, 29},
                          {30, 31, 32, 33, 34, 35, 36, 37, 38, 39},
                          {40, 41, 42, 43, 44, 45, 46, 47, 48, 49},
                          {50, 51, 52, 53, 54, 55, 56, 57, 58, 59},
                          {60, 61, 62, 63, 64, 65, 66, 67, 68, 69},
                          {70, 71, 72, 73, 74, 75, 76, 77, 78, 79} };

    imprime_matriz(8, 10, matriz);
    return 0;
}
```

Exercícios

- Escreva uma função que, dados dois números inteiros positivos, calcule e retorne o Máximo Divisor Comum (MDC) entre eles.
- Escreva uma função que, dados dois números inteiros positivos, calcule e retorne o Mínimo Múltiplo Comum (MMC) entre eles.
- Escreva um função que, dados um número real x e um número inteiro y , calcule e retorne o valor de x^y .
- Escreva um função que, dado um número inteiro positivo, imprima o número invertido. Por exemplo, se o número 3248700 for fornecido para função, ela deve imprimir 0078423.
- Escreva um função que, dado um número inteiro $n > 1$, imprima a fatoração em números primos de n . Por exemplo, se o número 936936 for fornecido para função, ela deve imprimir:

$$936936 = 1 \times 2^3 \times 3^2 \times 7 \times 11 \times 13^2$$

Exercícios

- Escreva um função que, dado um número inteiro positivo n e um vetor de n números inteiros, verifique se o vetor está ordenado de forma crescente.
- Escreva um função que, dado um número inteiro positivo n e um vetor de n números inteiros, compute o número de inversões do vetor. Dado um vetor π , dizemos que o par de elementos (π_i, π_j) é uma inversão se $i < j$ e $\pi_i > \pi_j$. Por exemplo, o vetor `[2 3 5 1 4]` contém 4 inversões: $(2, 1)$, $(3, 1)$, $(5, 1)$ e $(5, 4)$.
- Escreva uma função que calcule o tamanho de uma string dada (equivalente a função `strlen` da biblioteca `string.h`).
- Escreva uma função que inverta a ordem dos caracteres de uma string dada.
- Escreva um função que, dado um número inteiro positivo n e uma matriz quadrada $n \times n$, verifique se a matriz é simétrica.

Mínimo Múltiplo Comum (MMC)

```
int mmc(int x, int y) {  
    int resultado = 1;  
  
    while ((resultado % x) || (resultado % y))  
        resultado++;  
  
    return resultado;  
}
```

Mínimo Múltiplo Comum (MMC)

```
int mmc(int x, int y) {  
    int resultado = x;  
  
    while (resultado % y)  
        resultado += x;  
  
    return resultado;  
}
```

Mínimo Múltiplo Comum (MMC)

```
int mmc(int x, int y) {  
    int aux, resultado;  
  
    if (x < y) {  
        aux = x;  
        x = y;  
        y = aux;  
    }  
  
    resultado = x;  
  
    while (resultado % y)  
        resultado += x;  
  
    return resultado;  
}
```

Vetor ordenado

```
int ordenado(int n, int vetor[]) {
    int i, ok = 1;

    for (i = 0; (i < n - 1) && ok; i++)
        if (vetor[i] > vetor[i + 1])
            ok = 0;

    return ok;
}
```

Vetor ordenado

```
int ordenado(int n, int vetor[]) {
    int i;

    for (i = 0; i < n - 1; i++)
        if (vetor[i] > vetor[i + 1])
            return 0;

    return 1;
}
```