Programação de Computadores
Funções

Neste tópico abordaremos a construções de funções e C, procurando entender o conceito de escopo de variáveis, os mecanismos de passagem de parâmetros e estimulando a modularização de programas.

Prof. Ciro Cirne Trindade

Por quê usar funções?

- Evitar a repetição de um conjunto de instruções que ocorre várias vezes no programa
- Modularizar o programa
- Reaproveitamento de código

Estruturas das Funções em C

```
tipo nome-da-função([lista-de-parâmetros]) {
   corpo da função
}
```

- **tipo**: especifica o tipo do valor que a função retorna
- **lista-de-parâmetros**: uma lista separada por vírgulas das variáveis que recebem os valores dos argumentos passados quando a função é chamada
- **corpo da função**: comandos delimitados por chaves { }

Exemplo

```
#include <stdio.h>

void linha(void); //protótipo da função linha()

int main() {
    linha();
    printf("\n\t\t\t\tUm programa em C\n");
    linha();
    return 0;
}
void linha() {
    int i;
    for (i = 1; i <= 80; i++)
        printf("-");
}
```

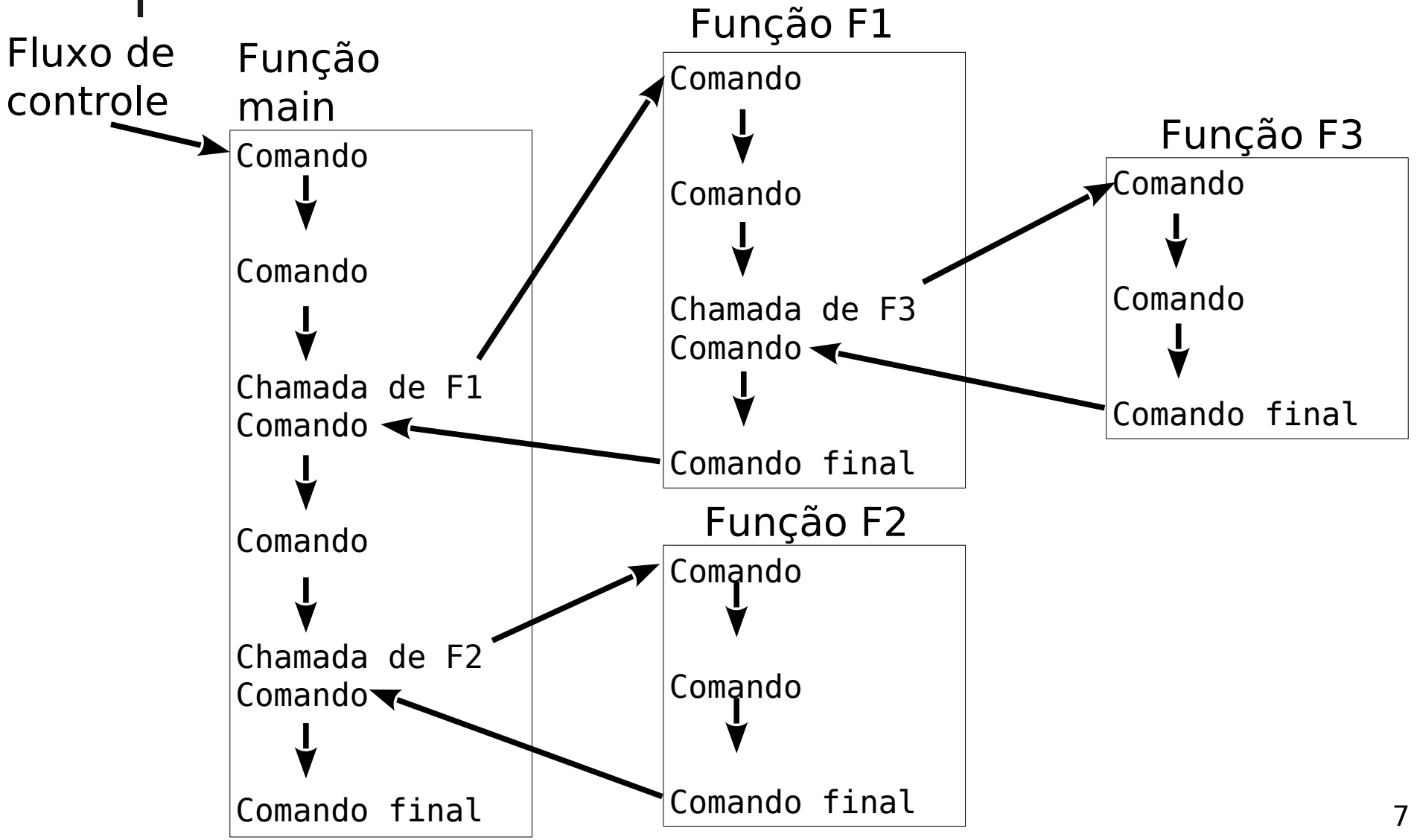
Protótipos de funções

- O protótipo de uma função informa ao compilador o tipo e o nome da função, além do número, ordem e tipos de seus argumentos
- O compilador usa protótipos de funções para validar as chamadas de funções
- **`void`** é usada na lista de parâmetros do protótipo para indicar uma lista de parâmetros vazia

Chamando Funções

- Do mesmo modo que chamamos uma função da biblioteca C (`printf()`, `scanf()`, etc.) chamamos nossas próprias funções como `linha()`
- Os parênteses que seguem o nome são necessários para que o compilador possa diferenciar a chamada a uma função de seu endereço de memória

Chamando funções



Variáveis Locais

- Variáveis que são declaradas dentro de uma função e são conhecidas somente dentro do seu próprio bloco
- Um bloco começa quando o compilador encontra uma chave de abertura ({) e termina quando o compilador encontra uma chave de fechamento (})
- Variáveis locais existem apenas durante a execução do bloco de instruções onde estão declaradas
  - Os blocos aonde uma variável é conhecida, chama-se **escopo** da variável

Passando Argumentos para Funções

- O mecanismo usado p/ transmitir informações a uma função é chamado argumento
  - O argumento também é chamado de parâmetro formal da função
- O argumento é uma nova variável que armazena a informação passada p/ a função
  - Funciona exatamente como uma variável local
  - É criada quando a função inicia sua execução e destruída quando a função termina

Passando Argumentos para Funções

- Argumentos - chamada por valor
  - Em C, todos os argumentos de funções são passados “por valor”, exceto vetores
  - Isto significa que à função chamada é dada uma cópia dos valores dos argumentos, e ela cria outras variáveis temporárias para armazenar estes valores
  - A função chamada *não pode* alterar o valor de uma variável da função que chama; ela só pode alterar sua cópia temporária

Exemplo

```c
#include <stdio.h>
void dobro(int);

int main() {
    int n;
    printf("Informe um número inteiro: ");
    scanf("%d", &n);
    dobro(n);
    printf("Número informado %d\n", n);
    return 0;
}

void dobro(int x) {
    x = x * 2;
    printf("Dobro: %d\n", x);
}
```

Funções que Devolvem um Valor

- O comando **`return`**
  - Causa uma saída imediata da função onde ele está contido, ou seja, termina a função
  - Pode ser usado para devolver um valor
    - Todas as funções, **exceto** aquelas do tipo **`void`**, devolvem um valor
    - Este valor é explicitamente especificado pelo comando **`return`**
    - O comando **`return`** faz com que a chamada à função que contém o **`return`** seja substituída pelo valor que sucede este comando

Funções que Devolvem um Valor

- Desde que uma função não seja declarada como **`void`**, ela pode ser usada como um operando em qualquer expressão
  - `x = pow(y, 2);`
  - `if (max(x, y) > 100) printf("Maior");`
  - `for (ch = getchar(); isdigit(ch); ) ...;`
  - `printf("Valor absoluto de %d = %d", x, abs(x));`
- Entretanto, uma função não pode receber uma atribuição
  - `sqr(x) = 100;  // instrução errada`

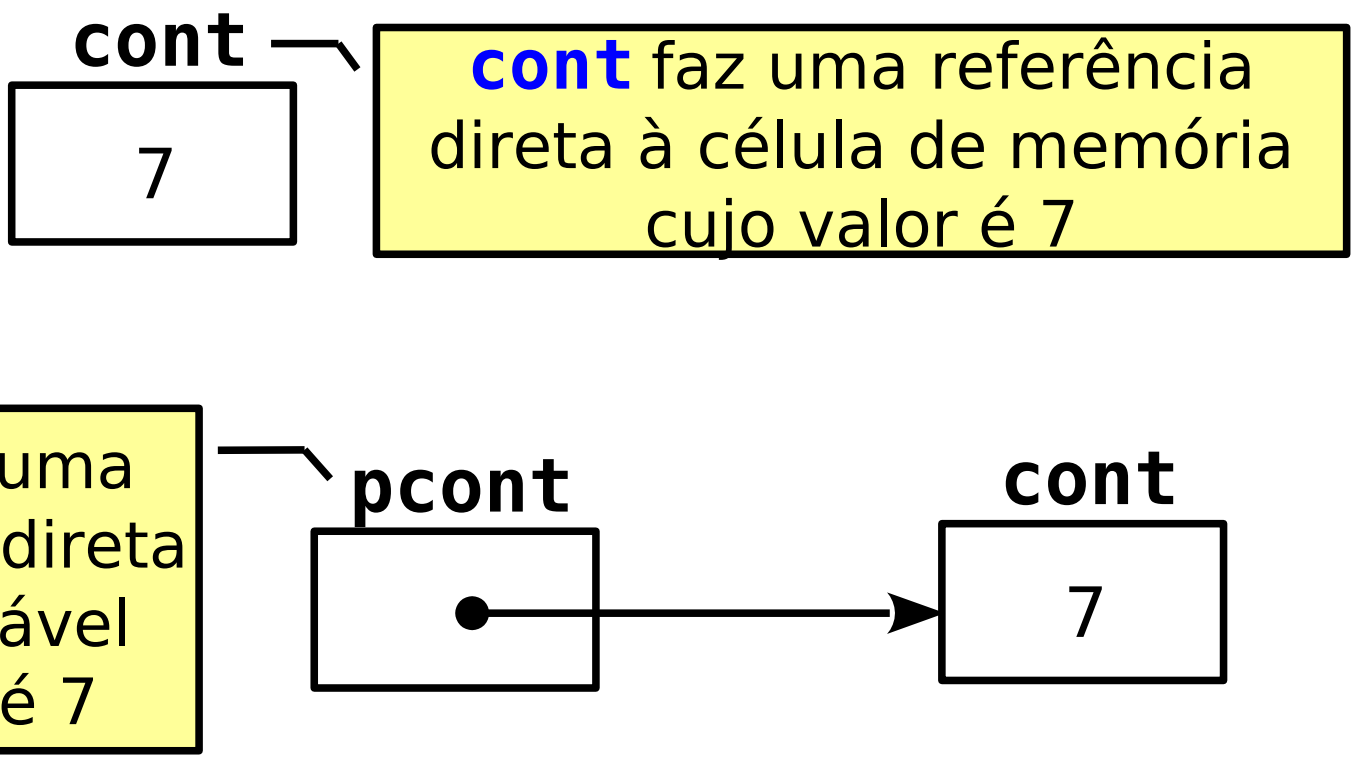
Exemplo

```c
#include <stdio.h>

int maior(int, int); //protótipo da função maior()

int main() {
    int n1, n2;
    printf("Digite dois números inteiros: ");
    scanf("%d %d", &n1, &n2);
    printf("O maior é: %d\n", maior(n1, n2));
    return 0;
}

int maior(int a, int b) {
    if (a > b)
        return a;
    return b;
}
```

Conceito de Ponteiro

- Ponteiros são variáveis especiais que armazenam endereços de memória
- Uma variável comum faz **referência direta** a um valor específico
- Um ponteiro contém o endereço de uma variável que contém um valor específico
- Ou seja, um ponteiro é uma **referência indireta** ao valor da variável cujo endereço ele armazena

Conceito de Ponteiro



Declarando Variáveis do Tipo Ponteiro

- Sintaxe:
  - `tipo * variável;`
- Exemplos:
  - `int * pcont;`
  - `float * px, * py;`
- Todo ponteiro tem um tipo específico que indica o tipo da variável para o qual ele aponta
  - Um ponteiro inteiro deve armazenar o endereço de memórias de variáveis inteiras

Inicializando Ponteiros

- Um ponteiro pode ser inicializado com:
  - 0: é convertido para um ponteiro do tipo apropriado
  - **`NULL`**: uma constante simbólica definida na arquivo de cabeçalho <stdio.h>
  - Um endereço de memória

Operadores de Ponteiros

- 2 operadores unários são usados:
  - **&**: obtém o endereço de memória de uma variável
  - ***: obtém o conteúdo do endereço de memória armazenado em um ponteiro
- Exemplo:

```
int cont = 7;
int * pcont = &cont;
printf("%d\n", *pcont);
```

Exemplo

```
#include<stdio.h>
int main() {
    int x = 10;// x é um inteiro
    int * px;  // px é um ponteiro p/ inteiro
    px = &x;  // px recebe o endereço de x
    printf("Endereço de x: %p\n", &x);
    printf("Valor de px: %p\n", px);
    printf("Valor de x: %d\n", x);
    printf("Valor do endereço armazenado em px: %d\n",
            *px);
    /* alterando dados */
    *px = 30;
    printf("Valor de x: %d\n", x);
    printf("Valor de px: %p\n", px);
    return 0;
}
```

Passagem de Parâmetros por Referência

- Por *default*, a passagem de parâmetros em C é feita por valor (com exceção de vetores e matrizes)
  - Uma cópia do valor do parâmetro é transmitida para a função
  - A função chamada **não pode** alterar o valor de uma variável da função que chama; ela só pode alterar sua cópia temporária
- A passagem de parâmetros por referência permite à função chamada alterar os valores das variáveis que foram passadas como parâmetro em sua chamada

Passagem de Parâmetros por Referência

- Na chamada à função, são transmitidos os endereços das variáveis
  - Usa-se o operador &
- Os parâmetros da função devem ser ponteiros

Exemplo

```
#include <stdio.h>

void troca(int * pa, int * pb) {
     int aux = *pa;
     *pa = *pb;
     *pb = aux;
}

int main() {
     int a = 10, b = 20;

     printf("a = %d, b = %d\n", a, b);
     troca(&a, &b);
     printf("a = %d, b = %d\n", a, b);
     return 0;
}
```

Exercícios

1) Criar uma função em C que receba como parâmetro o valor de um ângulo em radianos e converta-o para graus. Obs.: O parâmetro passado na chamada à função deve ser alterado para graus. Dica: graus = radianos*180/ π.

2) Criar uma função em C que receba um número inteiro $n > 0$ e devolve o número de dígitos de n e o primeiro dígito de n.

Vetores como Argumentos de Funções

- Chamamos a função com o nome do vetor/matriz sem nenhum índice
- Passamos o endereço do 1º elemento do vetor/matriz para a função
- Exemplo:

```
int main() {
  int vet[10];
  . . .
  fun(vet);
  . . .
}
```

Vetores como Argumentos de Funções

- Se a função recebe um vetor, o parâmetro formal pode ser um ponteiro, um vetor dimensionado, ou um vetor sem dimensão

```
void fun(int *a)        void fun(int a[10])     void fun(int a[])
{                       {                       {
  . . .                   . . .                   . . .
}                       }                       }
```

  - Esses 3 métodos dizem ao compilador que um ponteiro para um inteiro está sendo recebido
  - Passagem de parâmetros por referência

Matrizes como Argumentos de Funções

- Se a função recebe uma matriz bidimensional como argumento, o número de elementos da segunda dimensão (colunas) deve ser incluído no cabeçalho da função
- Exemplo: suponha **a** uma matriz 4 x 6

```
void fun(int a[4][6])
{
  ...
}
```

```
void fun(int a[][6])
{
    ...
}
```

```
void fun(int (*a)[6])
{
     ...
}
```

Matrizes como Argumentos de Funções

- A partir do C99 é possível definir a dimensão de uma matriz que é o parâmetro de uma função também através de parâmetros
- Exemplo: protótipo de função com uma matriz de dimensão variável como parâmetro

```
int soma2d(int lin, int col, int v[lin][col]);
```

  - Ou omitindo o nome dos parâmetros

```
int soma2d(int, int, int [*][*]);
```

Os parâmetros que definem a dimensão da matriz devem aparecer antes da matriz na lista de parâmetros da função

O Qualificador `const`

- Você pode impedir que o conteúdo do endereço de memória de um ponteiro seja alterado através do qualificador **`const`**
- Por exemplo:
  - **`void fun(const char * x);`**
    - **`x`** é não pode alterar o conteúdo do endereço que ele armazena
- Normalmente utiliza-se o **`const`** para evitar que o conteúdo de vetores sejam alterados na função, já que eles são passados automaticamente por referência

Passando Argumentos para `main()`

- A função `main()` também pode receber argumentos que são passados na chamada ao programa
- Forma geral:

```
int main(int argc, char * argv[]) {
   ...
}
```

  - `argc`: indica o número de argumentos
  - `argv`: vetor de string com os argumentos
    - O 1º argumento é o nome do programa

Exemplo: imprimindo os argumentos da `main()`

```
#include <stdio.h>
int main(int argc, char * argv[]) {
    int i;
    printf("Argumentos da main():\n");
    for(i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
    return 0;
}
```

Referências

- SCHILDT, Herbert. *C Completo e Total.* 3. ed., Makron Books, 1997.
- MIZRAHI, Victorine Viviane. Treinamento em linguagem C: [curso completo em um volume]. 2. ed. São Paulo: Pearson Prentice Hall, 2008.
- DEITEL, M.H.; DEITEL, P.J.. *Como Programar em C.* 2. ed., LTC, 1999.
- PRATA, Stephen. C Primer Plus. 6. ed., Addison Wesley, 2014.