Aula 03

Algoritmos e Programação II

Setembro / 2017

Prof. Mario Liziér
lizier@ufscar.br

Escopo

- Quando declaramos uma variável, precisamos defirnir seu **tipo** e **nome**
- Outro atributo que acabamos definindo em sua declaração é o **escopo**
- **O escopo** de uma variável determina o seu tempo de vida e os locais de acesso (**onde** e **quando** uma variável pode ser usada)
- Escopo global:
  - Variável declarada fora de todos os subprogramas (incluindo o *main*).
  - Seu tempo de vida é toda a execução do programa
  - Podemos acessar de qualquer subprograma
  - Desvantagem: torna o programa difícil de ser gerenciado, não conseguimos ter controle sobre quais subprogramas usam e/ou alteram as variáveis

Escopo Global

```c
#include <stdio.h>

int x = 3;

int main() {

    printf("%d\n", x);

    return 0;
}
```

```c
#include <stdio.h>

float x = 3.5;

float func(int a) {
    x += 10.0;
    if( a )
        return x + 1.0;
    else
        return x - 2.0;
}

int main() {

    x = 2.2;

    printf("%f %f\n", x, func(0) );

    return 0;
}
```

Escopo Local

- Definido pelos blocos { }
- As variáveis declaradas em um bloco { } só pode ser acessada de dentro deste bloco e é desalocada no seu término
- Acessamos na ordem de aninhamento dos blocos { } até o escopo global

```c
#include <stdio.h>

int x = 0;

int func(int a) {
    int x = 3;
    return x + a;
}

int main() {

    printf("%d\n", func(x) );

    {
        int x = 2;
        printf("%d\n", func(x) );
    }

    printf("%d\n", x);

    return 0;
}
```

Escopo Local

- Devemos evitar o uso de variáveis com nomes iguais!

```c
#include <stdio.h>

int main() {

    for(int i=0; i<3; i++) {
        printf("\n%d:\n", i);
        for(int i=0; i<3; i++) {
            printf("\n:%d\n", i);
            for(int i=0; i<3; i++)
                printf("%d ", i);
        }
    }

    return 0;
}
```

Subprogramas

- Quando invocamos (chamamos) um subprograma (função ou procedimento):
  1) A execução do programa atual é pausada
  2) Todos os valores passados como parâmetros são **copiados** para as variáveis definidas no subprograma
  3) O subprograma então é executado
     - No caso de uma função, o valor retornado (no *return*) é utilizado na chamada da função

```
a = funcao( ); // atribuição do retorno
outra( funcao( ) ); // uso do retorno como parâmetro
a = 3 + funcao( ) + 4; // uso do retorno em uma expressão
funcao( ); // descarte do retorno
```

  4) A execução do programa atual é continuada

Pilha de execução

- Os passos do slide anterior podem ser aplicados recursivamente, pois um subprograma pode chamar outro subprograma e assim por diante
- Para gerenciar estas invocações, o conceito de **pilha de execução** é utilizado!
- O topo da pilha é o espaço de trabalho (todas as variáveis locais (e parâmetros) e o ponteiro ou indicador de instrução)
- Quando uma invocação é feita, um novo espaço de trabalho é criado e colocado no topo da pilha
- Quando um subprograma termina, o espaço de trabalho do topo da pilha é desalocado e a execução continua com o novo topo
- Não existe limitação sobre qual subprorama podemos invocar, podendo ser o próprio subprograma! O que chamamos de **recursão**!

Recursão

- Em toda recusão precisamos ter:
  - Um ou mais casos bases:
    - Instância(s) resolvida(s) diretamente, sem precisar recorrer a uma nova instância (chamada)
  - Recorrência:
    - Instâncias que para serem resolvidas é necessário resolver instâncias “menores” (que convirjam para o(s) caso(s) base)

Recursões simples

- Fatorial

```c
int main(void)
{
    unsigned long n, f;

    printf("Digite n: ");
    scanf("%lu", &n);

    f = fatorial(n);
    printf("Resultado final = %lu\n", f);

    return 0;
}
```

```c
unsigned long fatorial(unsigned long n)
{
    unsigned long i, fat = 1;

  for(i = 2; i<=n; i++)
     fat = fat * i;

    return (fat);
}
```

```c
unsigned long fatorial(unsigned long n)
{
    if (n == 0)
            return 1;
    else
            return n * fatorial (n-1);
}
```

Recursões simples

- Fibonacci

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

Recursões simples

- Potência

$$x^n = \begin{cases} \frac{1}{x^{(-n)}} & \text{se } n < 0 \\ 1 & \text{se } n = 0 \\ x.x^{(n-1)} & \text{se } n > 0 \end{cases}$$

```
double pot(double x, int n) {
    if (n == 0) return 1;
    else if (n < 0)
            return 1/pot(x, -n);
        else
            return x*pot(x, n-1);
}
```

Recursões

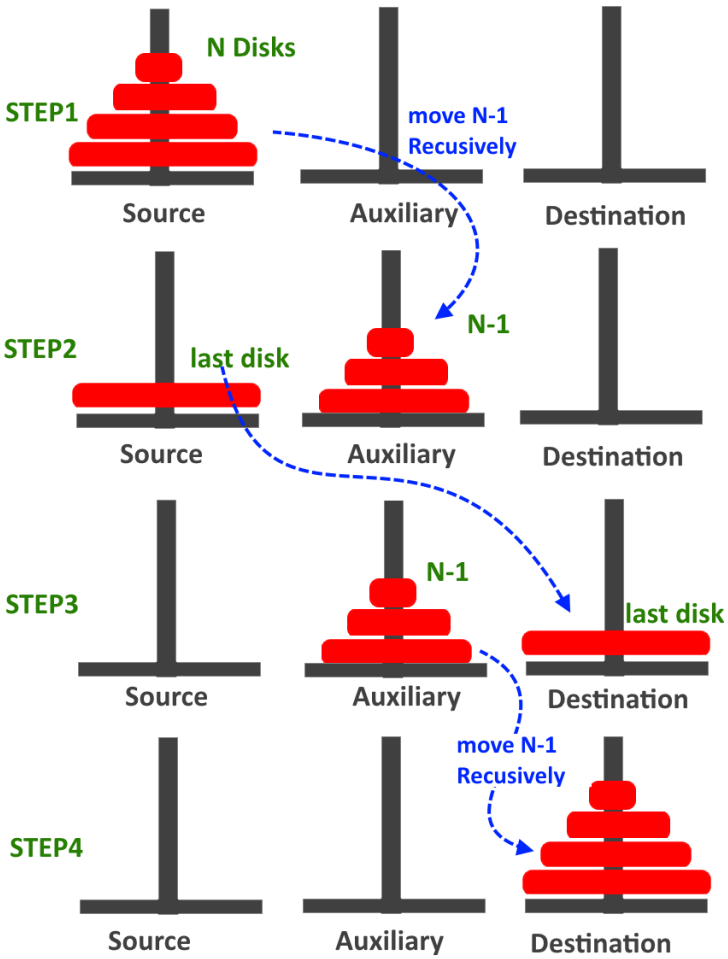
- É sempre mais simples usar um algoritmo recursivo?
- É sempre mais eficiente?

Exemplo – Soma e Maior elemento

- Faça uma rotina recursiva para somar um vetor de inteiros e encontrar o maior elemento
  - Receba o tamanho do vetor
  - Receba os elementos do vetor
  - Calcule recursivamente a soma dos elementos
  - Calcule recursivamente o valor do maior elemento
  - Imprima o valor da soma e do maior elemento

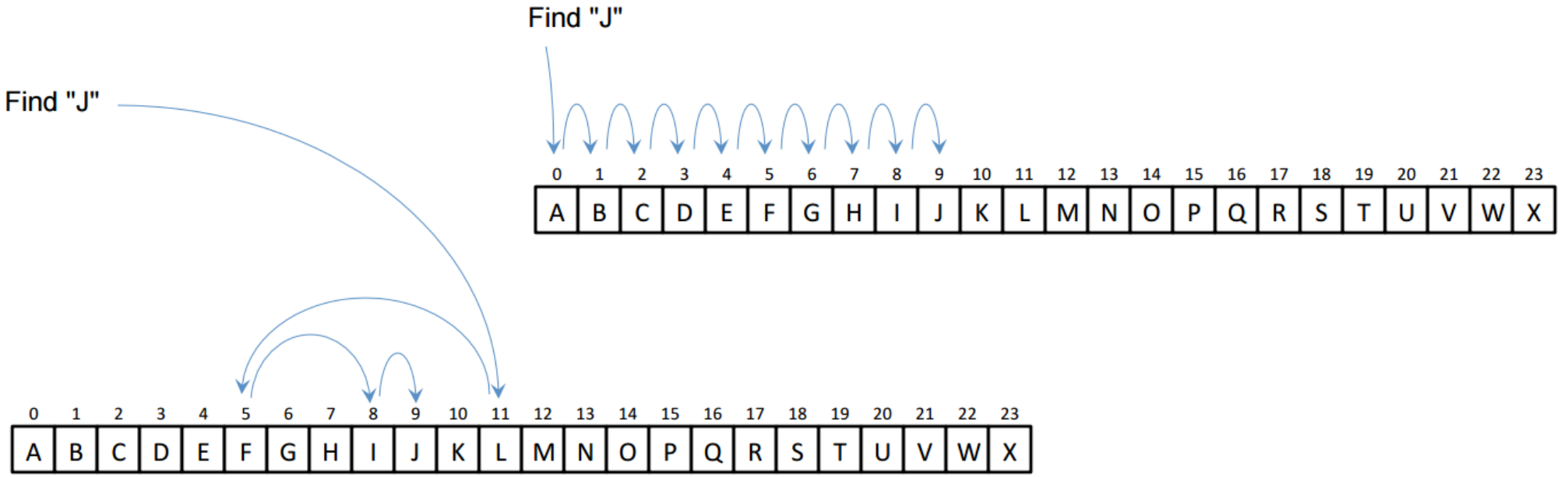
Exemplo - Hanoi

- Faça uma rotina resolver o problema da torre de Hanoi com ***n*** discos



Exemplo – Busca binária

- Faça uma rotina recursiva para procurar um determinado valor em um vetor de inteiros
  - Receba o tamanho do vetor
  - Receba os elementos do vetor (ou gere aleatóriamente e os imprima na tela)
  - Receba o valor a ser procurado
  - Imprima se o determinado valor está ou não presente no vetor



Exemplo – Busca binária

- Versão iterativa:

```
int binarySearch(int vet[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r-l)/2;

        if (vet[m] == x)
            return m;

        if (vet[m] < x)
            l = m + 1;
        else
            r = m - 1;
    }

    return -1;
}
```