CES-11

Algoritmos e Estruturas de Dados

Aula de revisão sobre programação

Carlos Forster
Carlos Alonso
Juliana Bezerra

CES-11

- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

CES-11

- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

Esclarecimento sobre notação

- Notação Algorítmica
  - Próxima à da Matemática e serve para explicar uma solução formalmente para uma pessoa.
  - Exemplo: atribuição ← i←i+1
  - Exemplo mais sofisticado: atribuição múltipla:
    - i,j←j,i (troca as variáveis i e j)
- Linguagem de Programação
  - Código para descrever um algoritmo para execução no computador.
  - É importante que seja legível por humano para que não precisemos reescrever tudo em notação algorítmica, principalmente quando se espera modificar o código com certa freqüência.

Tipos escalares primitivos

- Tipo **Inteiro**:
  - Domínio: números inteiros entre $-\infty$ e $+\infty$
  - Operações: + - * div(/) mod(%) = ≠ < ≤ > ≥
  - Exemplos: 7 div 3 = 2 7 mod 3 = 1
    - (em C) 7 / 3 = 2 7 % 3=1
- Tipo **Real**:
  - Domínio: números reais entre $-\infty$ e $+\infty$
  - Operações: + - * / = ≠ < ≤ > ≥
  - Arredondamento para inteiro: $\lfloor 3.14 \rfloor = 3$ $\lceil 3.14 \rceil = 4$
  - Lg x é o logaritmo de x na base 2.

Tipos escalares primitivos

- #include <limits.h> -- contem as constantes que descrevem os limites dos tipos inteiros da linguagem C.
  - #define SCHAR_MIN (-128)
  - #define SCHAR_MAX 127
  - #define UCHAR_MAX 255
  - #define INT_MAX 2147483647
  - #define INT_MIN (-INT_MAX-1)
  - #define UINT_MAX 0xffffffff
  - #define LONG_MAX 2147483647L
  - #define LONG_LONG_MAX 9223372036854775807LL

Tipos escalares primitivos

- #include<math.h> -- constantes e funções
  - #define M_E 2.7182818284590452354
  - #define M_PI 3.14159265358979323846
  - #define M_SQRT2 1.41421356237309504880
  - double sin (double); double cos (double); double tan (double);
  - double sinh (double); double cosh (double); double tanh (double);
  - double asin (double); double acos (double); double atan (double);
  - double atan2 (double, double);
  - double exp (double); double log (double); double log10 (double);
  - double pow (double, double);
  - double sqrt (double);
  - double ceil (double); double floor (double);
  - double fabs (double);

Tipos escalares primitivos

- Tipo **Lógico**:
  - Domínio: **Verdadeiro** e **Falso**
  - Operações: =, ≠, **and (&&)**, **or(||)**, **not(!)** e **xor(^)**
  - Os resultados das comparações são valores lógicos.

Tipo Lógico em C

- A linguagem C não define tipo lógico, mas utiliza inteiros. Um inteiro igual a zero corresponde a Falso e um inteiro qualquer diferente de zero corresponde a Verdadeiro.
- As operações lógicas da linguagem C retornam valores 0 para Falso ou 1 para Verdadeiro.
- Em C, os operadores lógicos fazem avaliação em "curto-circuito".

Tipos escalares primitivos

- Tipo Caractere
  - Domínio:
    - Dígitos decimais: '0', '1', ... , '9'
    - Letras: 'A', 'B', ... , 'Z', 'a', 'b', ... , 'z'
    - Sinais especiais: '.' , ',' , ':' , '+' , '-', '(' , ')','\n', '\t', ...
  - Operações: = ≠ (< ≤ > ≥ + - * / %)
  - Sucessor e Predecessor (Succ(x), Pred(x))
  - Número de ordem (Ord(x))
  - Caractere correspondente ao inteiro x (Chr(x))

Tipos escalares primitivos

- Os caracteres da linguagem C possuem valor de inteiro. Por isso, não estão definidas as operações de sucessor (x+1), predecessor (x-1), ordem e conversão para caractere ((char) x).
- Para a aritmética de caracteres, pode-se assumir que o código ASCII é utilizado (mas nem toda arquitetura utiliza esse código e há códigos em que as letras não aparecem em seqüência). Dessa forma é melhor, sempre que possível, utilizar as funções de ctype.h ao invés das comparações ≥ e ≤.

Tipos escalares primitivos

- #include <ctype.h>
  - int isalnum(int);
  - int isalpha(int);
  - int iscntrl(int);
  - int isdigit(int);
  - int isgraph(int);
  - int islower(int);
  - int isprint(int);
  - int ispunct(int);
  - int isspace(int);
  - int isupper(int);
  - int isxdigit(int);
  - int tolower(int);
  - int toupper(int);

Questões (ling C)

- Quando a expressão a>b>c é verdadeira?
  - Se a>b e c<1 ou se a<=b e c<0
  - Entretanto, não faz muito sentido comparar um valor lógico com um inteiro: (a<b)<c => Falso<c
- Qual a diferença entre
  - (getchar()=='s' || getchar()=='n')
  - (getchar()=='s' | getchar()=='n')
- E qual a diferença entre
  - x=3; k=x++;
  - x=3; k=++x;
- E a ordem alfabética vale?
  - 'b' > 'a'? 'B' > 'a'?

CES-11

- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

Tipos constituídos de uma linguagem

- Vetores:

```
Tipo_primitivo  v[30], w[50], x[200];
```

ou

```
typedef   int   vetor[30];
vetor   v1, v2, v3;
```

Inicialização:

int v[5]={5,4,3,2,1,};

int s[]={3,2,1};

Indexação: x=v[3] ou v[i+1]=x

v[i]: O resultado é um "L-Valor", ou seja, um valor com endereço em memória (na ling C)

Tipos constituídos de uma linguagem

- Matrizes:

```
Tipo_primitivo   m1[10][10][10],  m2[5][4];
```

ou

```
typedef   int  matriz[10, 10];
matriz   m3, m4;
```

Inicialização:

int x[][3]={{1,2,3},{4,5,6,},{7,8,9}};

float s[2][3]={{0,3,5},{2,1,4},};

Vetores e matrizes são chamados de variáveis indexadas

Estruturas homogêneas: contém elementos do mesmo tipo

Tipos constituídos de uma linguagem

- Cadeias de caracteres:

```
typedef   char  cadeia[15];
cadeia  nome, rua, aux;
```

As cadeias de caracteres na linguagem C funcionam como vetores do tipo char. As funções para manipulação das cadeias consideram que estas terminam com o caractere Nulo, de valor Zero ou '\0'.

```
#include <string.h>
    size_t strlen (const char*);
    char* strcat (char*, const char*);
    int strcmp (const char*, const char*);
    char* strcpy (char*, const char*);
```

Tipos constituídos de uma linguagem

- Estruturas simples (registro):

```
struct  funcionario {
   char  nome[30], endereco[30], setor[15];
   char  sexo, estCivil;
   int  idade;
};

struct funcionario  f1, f2, f3, empregados[200];
. . .
empregados[1] = f1;
f2.sexo = 'M';
strcpy (empregados[3].nome, "José da Silva");
```

Estruturas heterogêneas: podem agrupar elementos de vários tipos

Tipos constituídos de uma linguagem

- Outros:
  - Estruturas de campos alternativos (UNION)
  - Tipos enumerativos (ENUM)

```
enum numcor {AZUL=1,VERDE,AMARELO};

union u_cor {
  struct s_cor {
   unsigned char r, g, b;
  } uc_cor;
  enum numcor i_cor; };
```

Atribuição de Registros

```
typedef struct cadeia {
    char s[15];
};

cadeia help={"Auxilio"};
cadeia edit={"Editar"};

main()
{   cadeia k;
    k=help;
    printf("%s\n",k.s);
    k=edit;
    printf("%s\n",k.s);
    k.s[0]='X';
    printf("%s\n",k.s);
    printf("%s\n",edit.s);
}
```

Observe que não houve necessidade de utilizar strcpy neste exemplo porque as cadeias estão dentro de registros e a atribuição de registro já copia os vetores dentro dele

Saída no console:
Auxilio
Editar
Xditar
Editar

Estruturas *versus* implementações

Listas lineares
Pilhas
Filas
Árvores
Grafos
etc.

⟺

Variáveis indexadas
Ponteiros

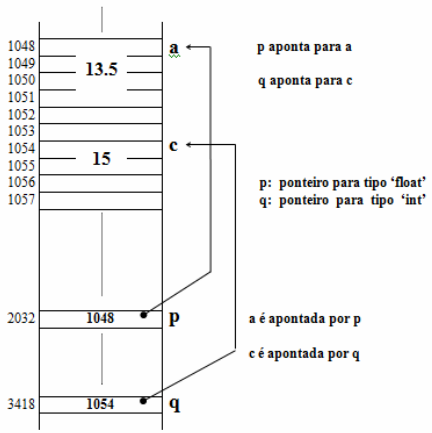
CES-11

- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

Ponteiros

- *Ponteiros* (ou *apontadores*) são variáveis que armazenam endereços de outras variáveis.
- No exemplo ao lado, `p` e `q` são ponteiros.
- Declarações:

```
float a; int c;
float *p; int *q;
p = &a; q = &c;
```

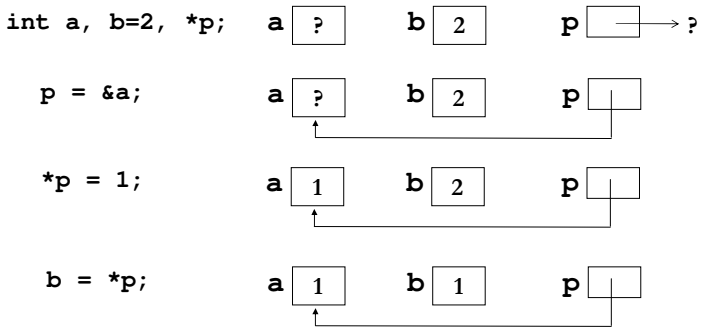


Ponteiros

- Principais utilidades de ponteiros:
  - Passagem de parâmetros por referência, em sub-programação
  - Alocação dinâmica de variáveis indexadas
  - Encadeamento de estruturas
  - Modificação direta de valores na memória

Ponteiros: notação

- Se `p` é um ponteiro, `*p` é o valor da variável apontada por `p`.
- Se `a` é uma variável, `&a` é o seu endereço.
- Exemplos:



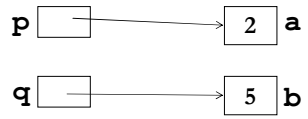
Ponteiros: exemplo

- Sejam as declarações abaixo:

  `int a=2, b=5, *p=&a, *q=&b;`

  `*q=&b` ou `q=&b` ???

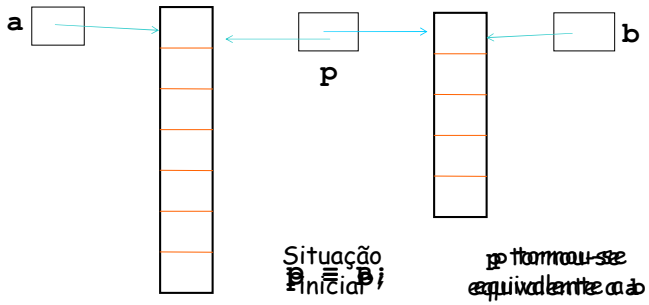
- A inicialização é de `p` e `q`, não de `*p` e `*q`:



Ponteiros e variáveis indexadas

- Sejam as declarações abaixo:

  `int a[7], *p, b[5];`



As atribuições `a = p` e `b = p` são proibidas!

Outras semelhanças

- Ponteiros podem ter índices, e variáveis indexadas admitem o operador unário '*'.
- Por exemplo, suponha as declarações abaixo:
  `int i, a[50], *p;`
  - `a[i]` é equivalente a `*(a+i)`
  - `*(p+i)` é equivalente a `p[i]`
- `a` contém o endereço de `a[0]`:
  - `p = a` equivale a `p = &a[0]`
  - `p = a+1` equivale a `p = &a[1]`

Qual é a diferença, então?

- Constante *versus* variável:
  - `a` é o endereço inicial de um vetor estático: seu valor não pode ser alterado
  - `p` é uma variável: seu conteúdo pode mudar
- Atribuições:
  - `p = &i` é permitido
  - `a = &i` não é permitido
- Endereços na memória:
  - `a[1]` tem sempre o mesmo endereço
  - `p[1]` pode variar de endereço

Aritmética de Apontadores

- A linguagem C permite certo conjunto de operações com apontadores. Essas operações não são comuns em linguagens de nível mais alto e devem ser evitadas.
- Sugere-se limitar a usar &a, *p, p[i], p++ e p--
- A comparação de apontadores p==q pode não funcionar em algumas plataformas (dois endereços diferentes podem representar o mesmo local na memória em algumas máquinas)
- O valor numérico do endereço pode ser obtido através da transformação em inteiro:
  - endereco=(unsigned long) p;
- Veja que endereco+1 pode ser diferente de p+1 (quando?)

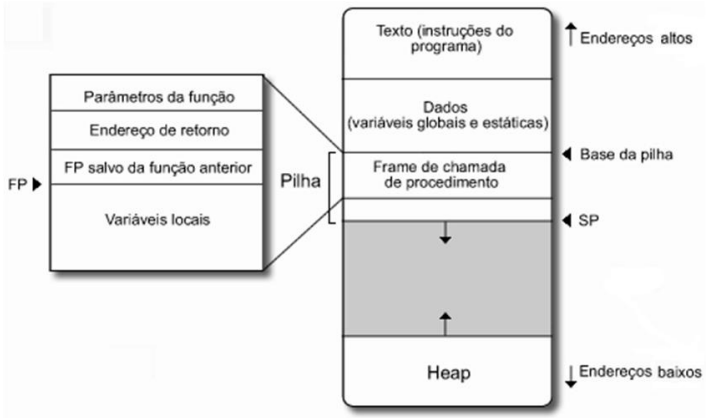
CES-11

- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

Alocação estática *versus* dinâmica

- Variáveis estáticas: têm endereço determinado em *tempo de compilação*
  - São previstas antes da compilação do programa
  - Ocupam uma área de dados do programa, determinada na compilação
  - Existem durante toda a execução do programa
- Variáveis dinâmicas: têm endereço determinado em *tempo de execução*
  - São alocadas de uma área extra da memória, chamada *heap*, através de funções específicas (`malloc`, `new`, etc.)
  - Sua eventual existência depende do programa, e seu endereço precisa ser armazenado em outra variável
  - Exige uma política de administração da memória

Pilha de execução



Alocação dinâmica de memória

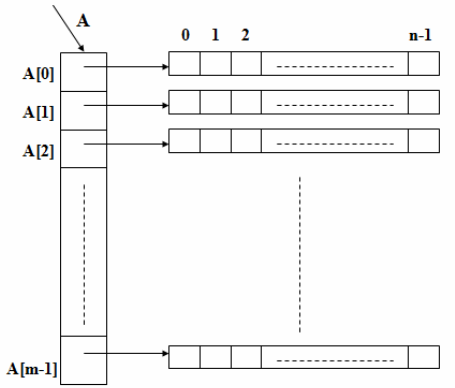
- Muitas vezes, é conveniente alocar espaço para uma variável indexada apenas em tempo de execução.
- Nesse caso, essa variável deve ser inicialmente alocada como ponteiro.
- Durante a execução do programa, o espaço de memória necessário para essa variável pode ser alocado através da função **`malloc`**.
- O espaço alocado **deve** ser reaproveitado utilizando-se a função **`free`** para liberá-lo.

Exemplo

```
typedef int *vetor;
void main () {
   int m, i; vetor A, B, C;
   printf("Tamanho dos vetores: ");
   scanf("%d",&m);
   A = (int *) malloc (m*sizeof(int));
   B = (int *) malloc (m*sizeof(int));
   C = (int *) malloc (m*sizeof(int));
   printf("Vetor A: ");
   for (i = 0; i < m; i++) scanf("%d",&A[i]);
   printf("Vetor B: ");
   for (i = 0; i < m; i++) scanf("%d",&B[i]);
   printf("Vetor C: ");
   for (i = 0; i < m; i++)
       C[i] = (A[i] > B[i])? A[i]: B[i];
   for (i = 0; i < m; i++) printf("%d",C[i]);
   free(A); free(B); free(C);
}
```

Alocação dinâmica de matrizes

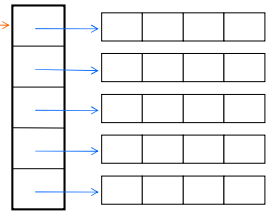
- Uma matriz também pode ser alocada em tempo de execução, de modo análogo aos vetores.
- Exemplo: matriz m x n.



Gasta-se mais espaço: um ponteiro para cada linha

Exemplo

```
typedef int *vetor;
typedef vetor *matriz;
void main () {
   int m, n, i, j; matriz A;
   printf("Dimensoes da matriz: "); scanf("%d%d",&m,&n);
   A = (vetor *) malloc (m * sizeof(vetor));
   for (i = 0; i < m; i++)
        A[i] = (int *) malloc (n * sizeof(int));
   printf("Elementos da matriz:");
   for (i = 0; i < m; i++) {
        printf("Linha %d ", i);
        for (j = 0; j < n; j++) scanf("%d",&A[i][j]);
   }
   for (i = 0; i < m; i++)
        free(A[i]);
   free(A);
}
```



Dimensões da matriz:

m 5 n 4

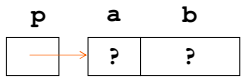
CES-11

- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

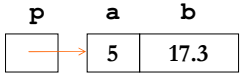
Encadeamento de estruturas

- Considere o código abaixo:

```
struct st {int a; float b}; st *p;
p = (st *) malloc (sizeof(st));
```




```
(*p).a = 5;  (*p).b = 17.3;
```

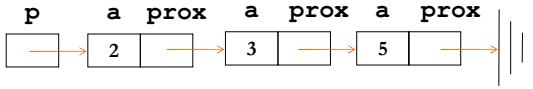


- Código equivalente às atribuições acima:

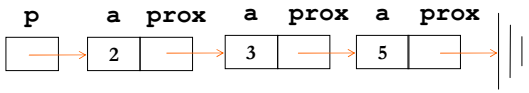
```
p->a = 5;  p->b = 17.3;
```

Outro exemplo

```
struct noh {int a; noh *prox};
noh *p;
p = (noh *) malloc (sizeof(noh));
p->a = 2;

p->prox = (noh *) malloc (sizeof(noh));
p->prox->a = 3;

p->prox->prox = (noh *) malloc (sizeof(noh));
p->prox->prox->a = 5;

p->prox->prox->prox = NULL;
```



Continuando



- Escrita do campo a de todos os nós:

```
noh *q;
for (q=p; q!=NULL; q=q->prox)
            printf("%d",q->a);
```

- Acesso ao campo a do último nó:

```
p->prox->prox->a
```
← mais simples

ou

```
(*(*(*p).prox).prox).a
```

Encadeamento de estruturas

- Baseia-se na utilização de variáveis ponteiros
- Proporciona muitas alternativas para estruturas de dados
- É usado em listas lineares, árvores e grafos

CES-11

- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

Passagem de parâmetros

- Declaração de funções:

Tipo *Nome_de_função* (*Lista_de_parâmetros*) {
  Corpo_de_função
}

  - Funções que não retornam valores são do tipo **void**
  - A lista de parâmetros pode ser vazia ou não
  - Parâmetros sempre são alocados dinamicamente, e recebem os valores que lhe são passados na chamada

Duas formas de passagem: por valor ou por referência

Passagem de parâmetros

- Passagem por **valor**

```
void ff (int a) {
      a += 1;
      printf ("Durante ff: a = %d \n", a);
}

void main ( ) {
      int a = 5;
      printf ("Antes de ff: a = %d \n", a);
      ff (a);
      printf ("Depois de ff: a = %d \n", a);
}
```

a 6

a 5

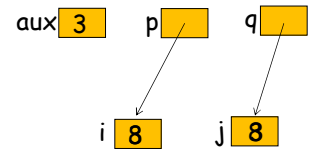
```
Antes de ff: a = 5
Durante ff: a = 6
Depois de ff: a = 5
```

← outra variável!

Passagem de parâmetros

- Passagem por **referência**

```
void trocar (int *p, int *q) {
   int aux;
   aux = *p; *p = *q; *q = aux;
}

void main ( ) {
   int i = 3, j = 8;
   printf ("Antes: i = %d, j = %d \n", i, j);
   trocar (&i, &j);
   printf ("Depois: i = %d, j = %d", i, j);
}
```




```
Antes: i = 3, j = 8
Depois: i = 8, j = 3
```

Outra vantagem: economia de memória ao se trabalhar com grandes estruturas

Passagem de parâmetros

- Passagem por **referência**
  - *Variável indexada como parâmetro*

```
#include <stdio.h>
void alterar (int B[]) {
   B[1] = 5;
   B[3] = 5;
}
void main ( ) {
   int i, j, A[10] = {0};
   //imprimir vetor A
   alterar(A);
   //imprimir vetor A
   alterar(&A[4]);
   //imprimir vetor A
 }
```

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

0	1	2	3	4	5	6	7	8	9
0	5	0	5	0	0	0	0	0	0

0	1	2	3	4	5	6	7	8	9
0	5	0	5	0	5	0	5	0	0

CES-11

- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

Recursividade

- Uma função é recursiva se fizer alguma chamada a si mesma diretamente ou através de uma outra função.
- Ex1: soma dos n primeiros números naturais

```
int soma (int n) {
   int i, resultado = 0;
   for (i=1; i<=n; i++)
      resultado = resultado + i;
   return resultado;
}
```

Mais elegante!

```
int somaRecursiva (int n) {
   if (n==1) return 1;
   return n + somaRecursiva(n-1);
}
```

Cuidado com *loop* infinito

Recursividade

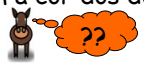
- Projeto de um algoritmo recursivo:
  - Procurar casos básicos em que a solução é conhecida sem esforço:
    - Em geral, quanto mais simples é o caso base, melhor.
    - No caso da soma da PA, poderíamos ter utilizado o caso base n=0 (a soma seria 0, o elemento neutro), mas utilizamos n=1, cuja soma é 1.
  - Agora pegamos o problema genérico de tamanho n e, supondo que sabemos resolver problemas menores, o resolvemos através da combinação de soluções desses problemas menores.
    - No caso da soma, utilizamos a soma de n-1 para calcular a soma de n.

Recursividade

- Veja que é fácil de provar por indução finita que o programa está correto.
- Mas há cuidados a se tomar, demonstramos o seguinte pelo princípio da indução finita:

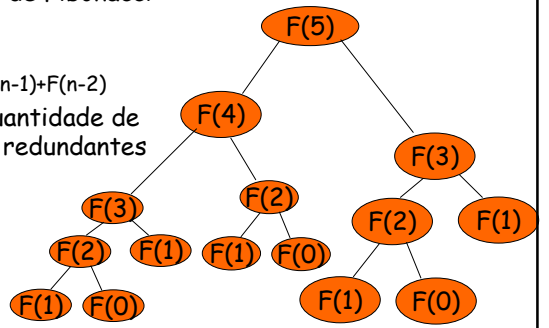
  "Em qualquer conjunto de cavalos, todos cavalos tem a mesma cor."
  - Num conjunto de 1 cavalo, todos tem a mesma cor.
  - Num conjunto de N cavalos, separamos N-1 e assumimos que tem a mesma cor, retiramos um cavalo do conjunto de N-1 e colocamos o que está sobrando, logo este também tem a cor dos demais.

Onde está o erro???



Recursividade

- Cuidado com o projeto eficiente do algoritmo recursivo:
  - Números de Fibonacci:
    - F(0)=1
    - F(1)=1
    - F(n)=F(n-1)+F(n-2)
  - Veja a quantidade de cálculos redundantes



Recursividade

- Ex2: cálculo de potência

$$A^n = \text{Power}(A, n) = \begin{cases} 1, & \text{se } n = 0 \\ A, & \text{se } n = 1 \\ A * \text{Power}(A, n-1), & \text{se } n > 1 \end{cases}$$

- Ex3: cálculo de fatorial de números positivos

$$\text{Fat}(n) = \begin{cases} -1 & \text{se } n < 0 \\ 1 & \text{se } n = 0 \text{ ou } n = 1 \\ n * \text{Fat}(n - 1) & \text{se } n > 1 \end{cases}$$

Recursividade

- Ex4: máximo divisor comum

$$\text{MDC}(m, n) = \begin{cases} m & \text{se } n = 0 \\ \text{MDC}(n, m \% n), & \text{se } n > 0 \end{cases}$$

42 = 2 * 3 * 7
30 = 2 * 3 * 5

MDC(42,30) = 6

m = n * q + r

MDC(42,30)	42 = 30 * 1 + 12
MDC(30,12)	30 = 12 * 2 + 6
MDC(12,6)	12 = 6 * 2 + 0
MDC(6,0)	Retorna 6

- Será que funciona se calcularmos MDC(30,42)?

Recursividade

- Ex4: máximo divisor comum

$$\text{MDC}(m, n) = \begin{cases} m & \text{se } n = 0 \\ \text{MDC}(n, m \% n), & \text{se } n > 0 \end{cases}$$

42 = 2 * 3 * 7
30 = 2 * 3 * 5

MDC(30,42) = 6

m = n * q + r

MDC(30,42)	30 = 42 * 0 + 30
MDC(42,30)	42 = 30 * 1 + 12
MDC(30,12)	30 = 12 * 2 + 6
MDC(12,6)	12 = 6 * 2 + 0
MDC(6,0)	Retorna 6

Recursividade

- Ex5: busca binária em vetor ordenado

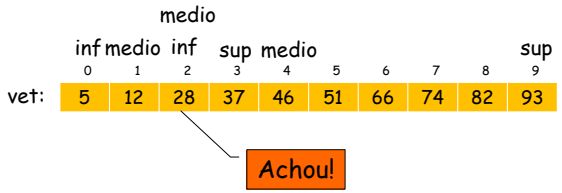
Procura (Elemento, Vetor, inf, sup) =

- -1, se Elemento < Vetor [inf] ou se Elemento > Vetor [sup]
- médio, se Elemento = Vetor [medio]
- Procura (Elemento, Vetor, inf, medio - 1), se Elemento < Vetor [medio]
- Procura (Elemento, Vetor, medio + 1, sup), se Elemento > Vetor [medio]

medio = (inf + sup) /2

Recursividade

- Ex5: busca binária em vetor ordenado
  - Procura(28,vet,0,9)



Recursividade

- Ex6: reconhecimento de cadeias de caracteres
  - Uma cadeia contendo apenas uma letra ou dígito é válida.
  - Se **α** é uma cadeia válida, então **(α)** também será.

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

typedef char cadeia[30];
cadeia cad;
int i;
bool erro;

void testarCadeia (void);
```

Contém `printf`

Contém `getche()` e `gets()`

Contém operações para *string*

Variáveis globais

Protótipo de função

Recursividade

- Ex6: reconhecimento de cadeia de caracteres

```
void main() {
       char c;
       printf ("Testar cadeia? (s/n): ");
       do c = getche ( );
       while (c!='s' && c!='n');
       while (c == 's') {
              clrscr (); printf ("Digite a cadeia: ");
              fflush (stdin); gets (cad);
              i = 0; erro=false;
              testarCadeia ();
              if (cad[i] != '\0') erro = true;
              if (erro) printf ("cadeia reprovada!");
              else printf ("cadeia valida!");
              printf ("\n\nTestar nova cadeia? (s/n): ");
              do c = getche ( );
              while (c!='s' && c!='n'); } }
```

Recursividade

- Ex6: reconhecimento de cadeia de caracteres

```
void testarCadeia () {
       if (isalpha(cad[i]) || isdigit(cad[i]))
              i++;
       else if (cad[i] == '(') {
              i++;
              testarCadeia ();
              if (erro==false && cad[i] == ')')
                     i++;
              else erro = true;
       }
       else erro = true;
}
```

Recursividade

- Ex6: reconhecimento de cadeia de caracteres
  - O código funciona, mas não é elegante.
  - Pontos fracos:
    - Não verifica se a cadeia excede 30 caracteres
    - **cad**, **i** e **erro** são variáveis globais
      - O ideal seria: **bool testarCadeia(cadeia,indice)**
    - Para que o código abaixo está na função `main`?

```
if (cad[i] != '\0') erro = true;
```

      - Gera erro em casos com 2 letras/números seguidos. Ex: aa, (bb)
      - Isso deveria ser escopo da função `testarCadeia`.

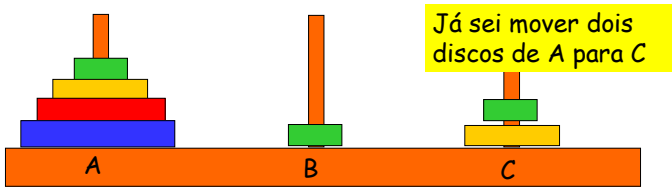
Recursividade

- Ex7: reconhecimento de expressões aritméticas
  - Uma expressão com apenas uma letra ou dígito é válida.
  - Se **α** e **β** são expressões válidas, então **(α+β)**, **(α-β)**, **(α*β)** e **(α/β)** também serão.
- Ex8: cadeias com **n** zeros iniciais **(n ≥ 0)** seguidos de **2n** um's
  - Uma cadeia vazia é válida.
  - Se **α** é uma cadeia válida, então 0**α**11 também é.

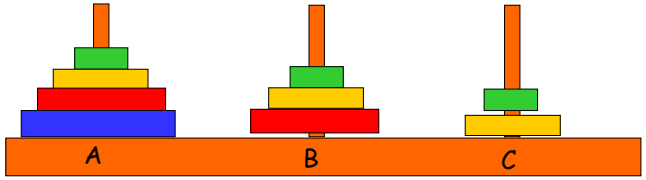
Torres de Hanoi

Solução para torres de Hanoi

- Regras:
  - Três colunas A, B e C
  - Os 64 discos estão todos na coluna A inicialmente
  - Devem estar todos na coluna C no final
  - Todos discos tem tamanho diferente
  - Um disco maior nunca pode estar sobre um menor



Torres de Hanoi



Já sei mover dois discos de A para C:
A->B, A->C, B->C

Como mover 3?

Mover 2 de A para C
Mover 1 de A para B
Mover 2 de C para B

Torres de Hanoi

```
void mover(int quantidade, char *de, char *para, char *por)
{
    if(quantidade==1) printf("%s->%s ",de,para);
    else
    {
       mover(quantidade-1, de, por, para);
       mover(1, de, para, por);
       mover(quantidade-1, por, para, de);
    }
}
main()
{
     mover(4,"A","C","B");
     getchar(); getchar();
}
```

A->B A->C B->C A->B
C->A C->B A->B A->C
B->C B->A C->A B->C
A->B A->C B->C

Fim