INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SUL-RIO-GRANDENSE

Estrutura de Dados

Prof. Silvana Teodoro

silvanateodoro@charqueadas.ifsul.edu.br

Vetores

Objetivos

- Revisar conceitos relacionados a utilização das **estruturas de dados homogêneas**, mais especificamente referente ao uso de **vetores,** dentro da linguagem C.

Estruturas de dados homogêneas

- São estruturas que permitem armazenar conjuntos de **dados** de um **mesmo tipo** (daí o nome "**homogêneas"**) em uma única variável. São também chamadas de **variáveis compostas homogêneas** ou **variáveis compostas indexadas**.
- Variável que constitui-se de um ***conjunto de posições de memória***, capaz de armazenar um ***certo número de valores*** de acordo com o número de posições de memória especificadas na declaração da mesma;
- **Cada posição** de memória é localizada na variável através de um ou mais **índices**.

Vetores e Matrizes

- Uma variável indexada através de um ***único índice*** é denominada ***vetor ou matriz unidimensional***;

- Uma variável indexada por ***dois índices*** é denominada ***matriz bidimensional***. As demais podem ser denominadas genericamente de ***matrizes multidimensionais***.

Vetores e Matrizes

- **Importante:** C não verifica o índice **[i]** usado, assim deve-se assumir valores dentro dos limites válidos;
- Se o programador não tiver ***atenção*** com os ***limites válidos*** para os índices, ele corre o risco de ter variáveis sobrescritas ou de ver o computador travar;
- O índice sempre se **inicia em 0** (zero).

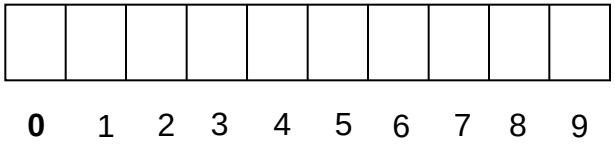
Vetores

- Declaração:

```
tipo nome_da_variável [tamanho];
```

- Exemplo:

```
int v[10];
```



Vetores

- Memória: vetor é alocado em **posições contíguas de memória.**
- Exemplo:
  - v = vetor de inteiros com 10 elementos
  - espaço de memória de v =

    10 x valores inteiros

Vetores

- Acesso: acesso a cada elemento é feito através de **indexação** da variável
- No exemplo anterior:

```
v[0] = 0;          /*  acessa o primeiro elemento de v     */
...
v[9] = 9;          /*  acessa o último elemento de v       */

v[10]  =10         /*  ERRADO (invasão de memória)         */
```

Vetores

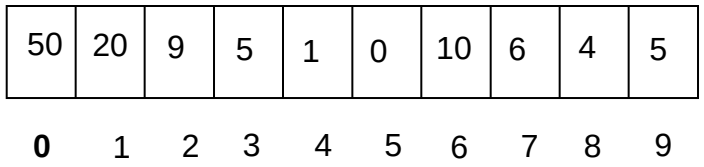
- Inicialização:

```
tipo  nome_vetor[tamanho] = {valor, valor, ...}
```

  - Se o **tamanho for omitido**, o vetor será dimensionado de acordo com o **número de elementos inicializados**.

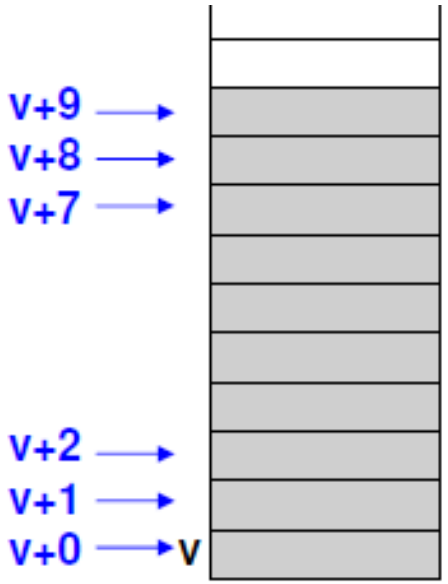
- Exemplo:

```
int vet[10]={50, 20, 9, 5, 1, 0, 10, 6, 4, 5};
int vet[]={50, 20, 9, 5, 1, 0, 10, 6, 4, 5};
```



Vetores

- Nome do vetor aponta para **endereço inicial**
- C permite **aritmética de ponteiros**:
  - v+0 : primeiro elemento de v
  - ...
  - v+9 : último elemento de v
- Concluindo:
  - &v[i] é equivalente a (v+i)
  - *(v+i) é equivalente a v[i]



Recursividade

Conceito

- Fundamental em Matemática e Ciência da Computação
  - Um **programa recursivo** é um programa que **chama a si mesmo**
  - Uma **função recursiva** é **definida em termos dela mesma**
- Exemplos:
  - Função fatorial, Árvore, etc...
- Conceito poderoso:
  - Define **conjuntos infinitos** com **comandos finitos**

Recursividade

- Fatorial(n) ou n!

  = n*(n-1)*(n-2)...*1 se n >0

  =1 se n==0

- Se formulamos:

  0! = 1

  1! = 1

  2! = 2 * 1

  3! = 3 * 2 * 1

  4! = 4 * 3 * 2 * 1

- Não é possível listar a fórmula para fatorial de cada inteiro (infinito...)

Recursividade

- A recursividade é uma estratégia que pode ser utilizada sempre que o cálculo de uma função para o valor n, pode ser descrita a partir do cálculo desta mesma função para o termo anterior (n-1).

Ex 4! = 4 * 3 * 2 * 1 isso é igual a 4 * 3!

Para todo n>0 verificamos que

n! = n* (n-1)!

Podemos definir:

0! = 1

1! = 1 * 0!

2! = 2 * 1!

3! = 3 * 2!

4! = 4 * 3!

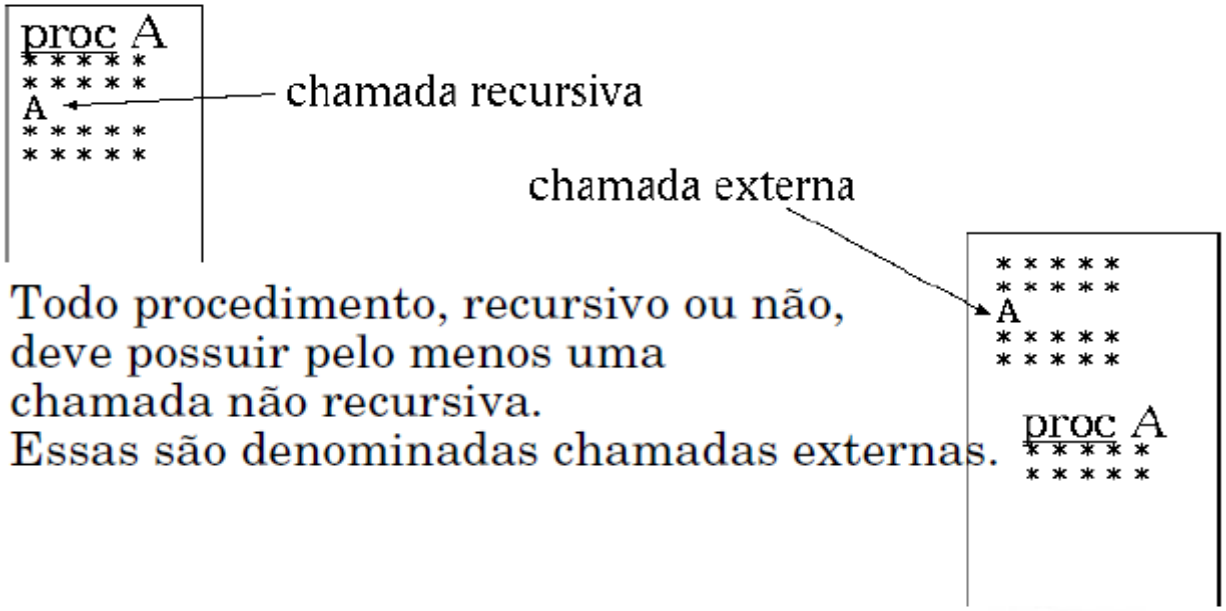
Exemplo: Fatorial

```
int fatorial(int n) {
    if (n<=0)
        return 1;
    else
        return n * fatorial(n-1);
}

int main() {
   int f;
   f = fatorial(4);
   printf("%d",f);
   getch();
}
```

Recursividade

- Definição: **dentro** do corpo de uma função, **chamar novamente a própria função**.
  - recursão **direta**: a função A chama a própria função A
  - recursão **indireta**: a função A chama uma função B que, por sua vez, chama A



Quando usar?

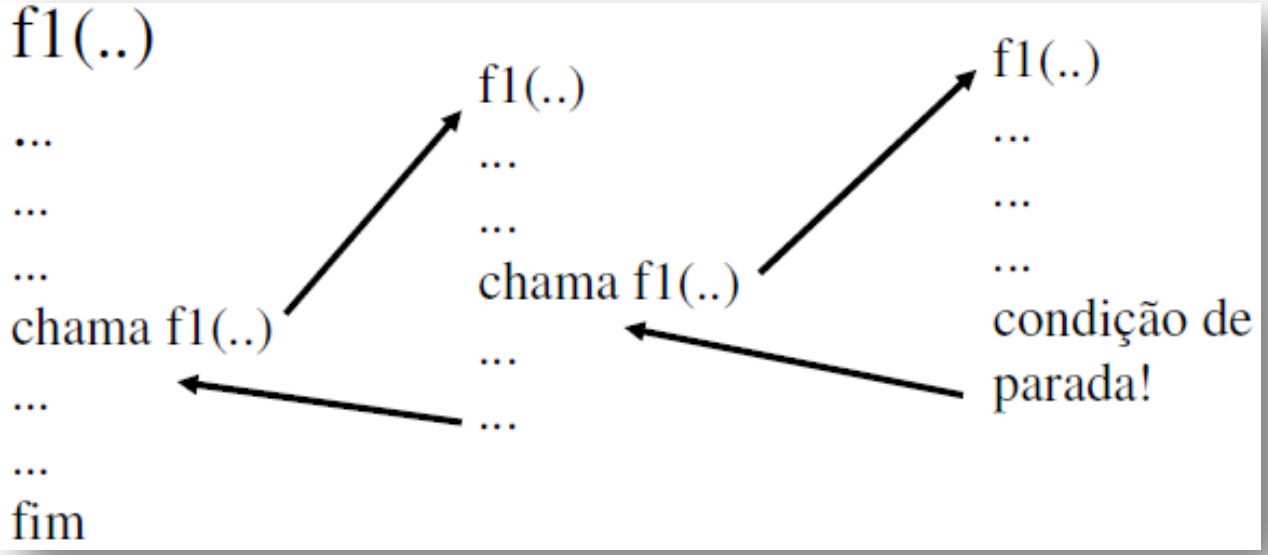
- A recursividade pode ser utilizada quando um **problema puder ser definido em termos de si próprio**.
- **Exemplo**: quando um objeto é colocado entre dois **espelhos** planos **paralelos** e frente a frente surge uma **imagem recursiva**, porque a imagem do objeto refletida num espelho passa a ser o objeto a ser refletido no outro espelho e, assim, sucessivamente.
- Uma função recursiva chama ela mesma, mas **com outros parâmetros**.

Algoritmo recursivo

- A idéia básica de um algoritmo recursivo consiste em **diminuir sucessivamente o problema em um problema menor ou mais simples**, até que o tamanho ou a simplicidade do problema reduzido permita **resolvê-lo de forma direta**, **sem recorrer a si mesmo.**
- Quando isso ocorre, diz-se que o algoritmo atingiu uma **condição de parada**, a qual deve estar presente em pelo menos um local dentro algoritmo.
- **Sem esta condição o algoritmo não pára** de chamar a si mesmo, até estourar a capacidade da pilha de execução, o que geralmente causa efeitos colaterais e até mesmo o término indesejável do programa.

Componentes do algoritmo recursivo

- **Condição de parada**, quando a parte do problema pode ser resolvida diretamente, sem chamar de novo a função recursiva.
- **Outros comandos** que resolvem uma parte do problema (chamando novamente a função recursiva);

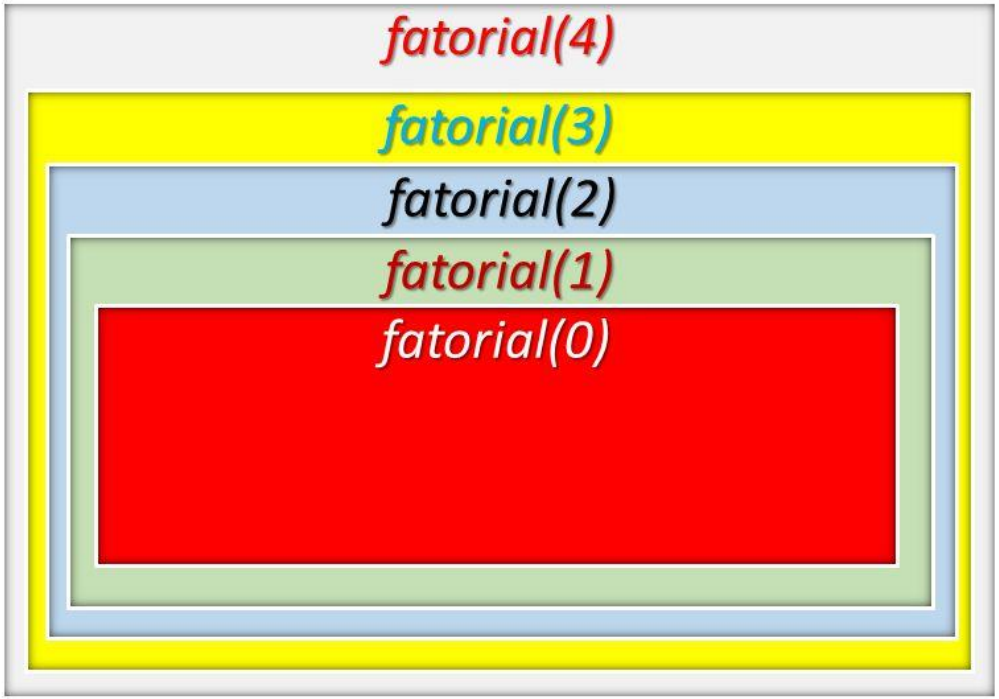


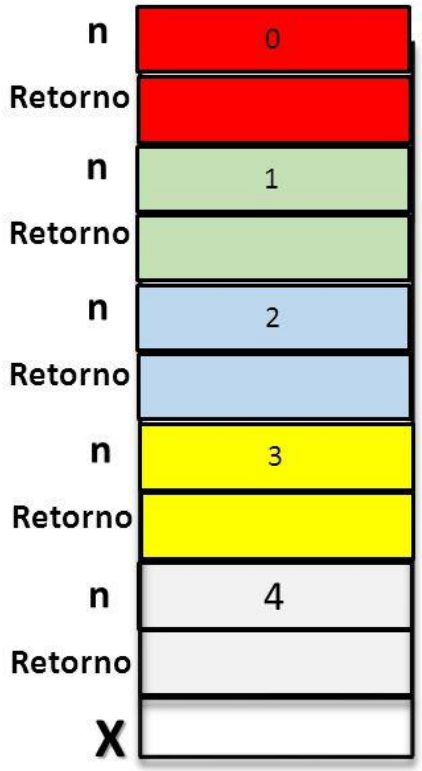
Pilha de execução

- Para cada chamada de uma função, **recursiva ou não**, os **parâmetros** e as **variáveis locais** são **empilhados** na **pilha de execução**.
- Internamente, quando qualquer chamada de função é feita dentro de um programa, é criado um **Registro de Ativação** na **Pilha de Execução** do programa
- O registro de ativação armazena os parâmetros e variáveis locais da função bem como o "**ponto de retorno**" no programa ou subprograma que chamou essa função.
- Ao **final da execução** dessa função, o **registro é desempilhado** e a **execução volta** ao subprograma que chamou a função

Exemplo: Fatorial

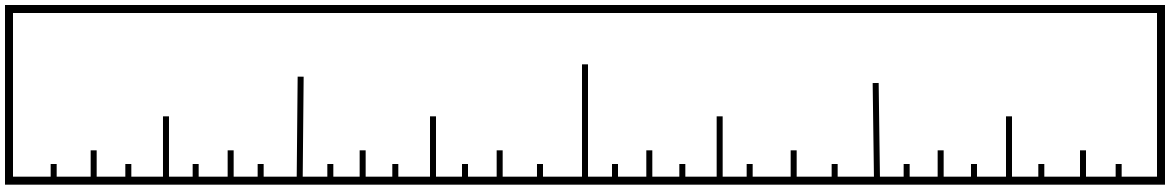
Recursão e a Pilha de Execução *(stack)*



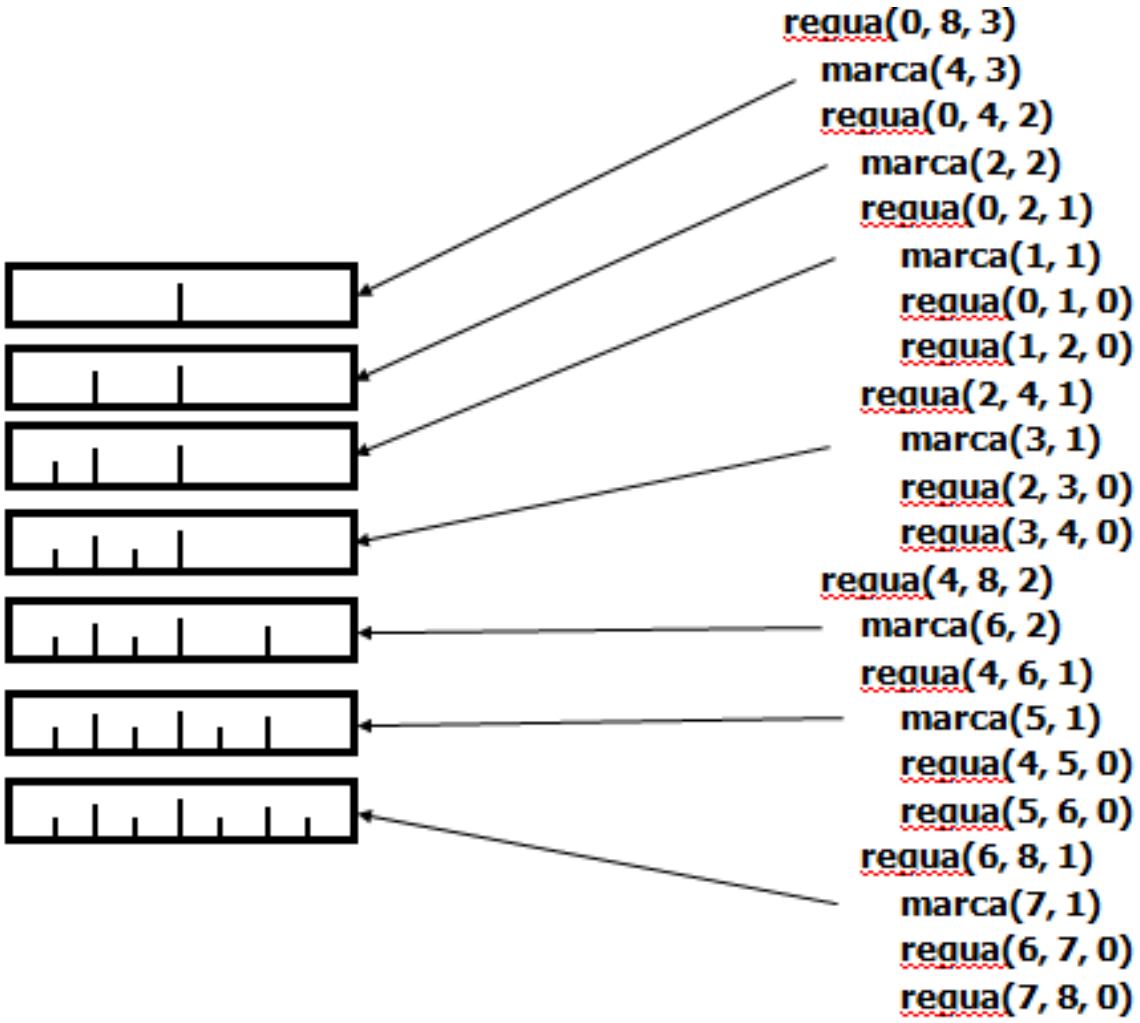


Exemplo: Régua

```
int regua(int l,int r,int h)
{
  int m;
  if ( h > 0 )
  {
    m = (l + r) / 2;
    marca(m, h);
    regua(l, m, h - 1);
    regua(m, r, h - 1);
  }
}
```



Exemplo: Régua



Desafio

- Analise e explique o que faz a função abaixo:

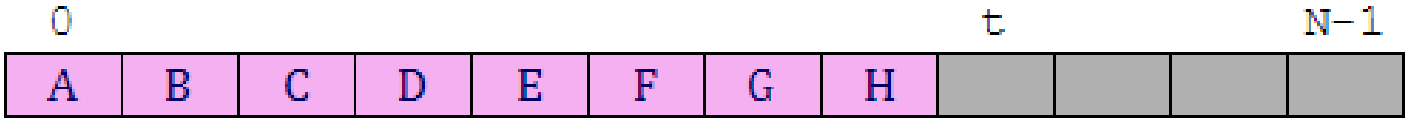
```
int f(int a, int b) { // considere a > b
  if (b == 0)
    return a;
  else
    return f(b, a % b);
}
```

Implementação de Pilha com Vetor

- Digamos que a parte do vetor ocupada pela pilha é

  pilha[0..t-1]

- O índice t indica a primeira posição vaga da pilha e t-1 é o índice do *topo* da pilha. A pilha está *vazia* se t vale 0 e *cheia* se t vale N. No exemplo da figura, os caracteres A, B, … , H foram inseridos na pilha nessa ordem:



Implementação de Pilha com Vetor

- Para *remover*, ou *tirar*, um elemento da pilha , ***desempilhar* (*pop*)**:

```
x = pilha[--t];
```

  - Isso equivale ao par de instruções `t -= 1;` `x = pilha[t];`, nessa ordem. É claro que você só deve desempilhar se tiver certeza de que a pilha não está vazia.

- Para *inserir*, ou *colocar*, um objeto y na pilha , ***empilhar* (*push*)**:

```
pilha[t++] = y;
```

  - Isso equivale ao par de instruções `pilha[t] = y;` `t += 1;`, nessa ordem. Antes de empilhar, certifique-se de que a pilha não está cheia, para evitar um *transbordamento* (*overflow*).

Implementação de Pilha com Vetor

```
char desempilha (void) {
    return pilha[--t];
}

void empilha (char y) {
    pilha[t++] = y;
}
```

Implementação de Pilha com Vetor

Estamos supondo aqui que as variáveis pilha e t são *globais*, isto é, foram declaradas fora do código das funções.

#define N 100

char pilha[N];

int t;

```
void criapilha (void) {
   t = 0;
}

void empilha (char y) {
   pilha[t++] = y;
}

char desempilha (void) {
   return pilha[--t];
}

int pilhavazia (void) {
   return t <= 0;
}
```

Exercício

1) Escreva um algoritmo que use uma pilha para inverter a ordem das letras de cada palavra de uma string, preservando a ordem das palavras. Por exemplo, para a string ESTE EXERCICIO E MUITO FACIL o resultado deve ser ETSE OICICREXE E OTIUM LICAF.

2) Escreva um algoritmo que use uma pilha para verificar se uma expressão está correta.

(( ) [( )])

([ )]

Atividade!

- Na notação usual de expressões aritméticas, os operadores são escritos *entre* os operandos; por isso, a notação é chamada *infixa*. Na notação *posfixa*, os operadores são escritos *depois* dos operandos.

infixa	posfixa
(A+B*C)	ABC*+
(A*(B+C)/D-E)	ABC+*D/E-
(A+B*(C-D*(E-F)-G*H)-I*3)	ABCDEF-*-GH*-*+I3*-
(A+B*C/D*E-F)	ABC*D/E*+F-
(A+B+C*D-E*F*G)	AB+CD*+EF*G*-
(A+(B-(C+(D-(E+F)))))	ABCDEF+-+-+
(A*(B+(C*(D+(E*(F+G))))))	ABCDEFG+*+*+*

Atividade!

- Note que os operandos (A, B, C, etc.) aparecem na mesma ordem na expressão infixa e na correspondente expressão posfixa.

- Note também que a notação posfixa *dispensa parênteses* e *regras de precedência* entre operadores (como a precedência de * sobre + por exemplo), que são indispensáveis na notação infixa.

Atividade!

- Nosso problema: traduzir para notação posfixa a expressão infixa armazenada em uma string inf.

- Para simplificar nossa vida, vamos supor que a expressão inf é válida e contém apenas letras, parênteses esquerdos, parênteses direitos, e símbolos para as quatro operações aritméticas, todas as operações (em particular - e +) têm *dois* operandos, os nomes das variáveis têm apenas uma letra cada, a expressão inf está embrulhada em um par de parênteses (ou seja, o primeiro caractere é '(' e os dois últimos são ')' e '\0').

Atividade!

- O algoritmo lê a expressão inf caractere-a-caractere e usa uma pilha para fazer a tradução.
- Todo parêntese esquerdo é colocado na pilha. Ao encontrar um parêntese direito, o algoritmo desempilha tudo até o primeiro parêntese esquerdo inclusive.
- Ao encontrar um + ou um - , o algoritmo desempilha tudo até um parêntese esquerdo exclusive.
- Ao encontrar um * ou um / , o algoritmo desempilha tudo até um parêntese esquerdo ou um + ou um. Constantes e variáveis são transferidos diretamente de inf para a expressão posfixa.

Atividade!

- As variáveis pilha e t são globais.
- Vamos supor que o tamanho N da pilha é maior que o tamanho da string inf, e portanto não precisamos nos preocupar com pilha cheia.
- Como a expressão inf está embrulhada em parênteses, não precisamos nos preocupar com pilha vazia.

Atividade!

```
#define N 100
char pilha[N];
int t;

// Esta função recebe uma expressão infixa inf
// e devolve a correspondente expressão posfixa.

char *infixaParaPosfixa (char *inf) {
   char *posf;
   int i, j;

   n = strlen (inf);
   posf = mallocc ((n+1) * sizeof (char));
   criapilha ();
   empilha (inf[0]);        // empilha '('

   for (j = 0, i = 1; inf[i] != '\0'; ++i) {
      switch (inf[i]) {
         char x;
         case '(': empilha (inf[i]);
                   break;
         case ')': x = desempilha ();
                   while (x != '(') {
                      posf[j++] = x;
                      x = desempilha ();
                   }
                   break;
```

Atividade!

```
        case '+':
        case '-': x = desempilha ();
                  while (x != '(') {
                     posf[j++] = x;
                     x = desempilha ();
                  }
                  empilha (x);
                  empilha (inf[i])
                  break;
        case '*':
        case '/': x = desempilha ();
                  while (x != '(' && x != '+' && x != '-') {
                     posf[j++] = x;
                     x = desempilha ();
                  }
                  empilha (x);
                  empilha (inf[i]);
                  break;
        default:  posf[j++] = inf[i];
     }
  }
  posf[j] = '\0';
  return posf;
}
```

Atividade!

- Use a função infixaParaPosfixa para converter a expressão infixa (A+B)*D+E/(F+A*D)+C na expressão posfixa equivalente.

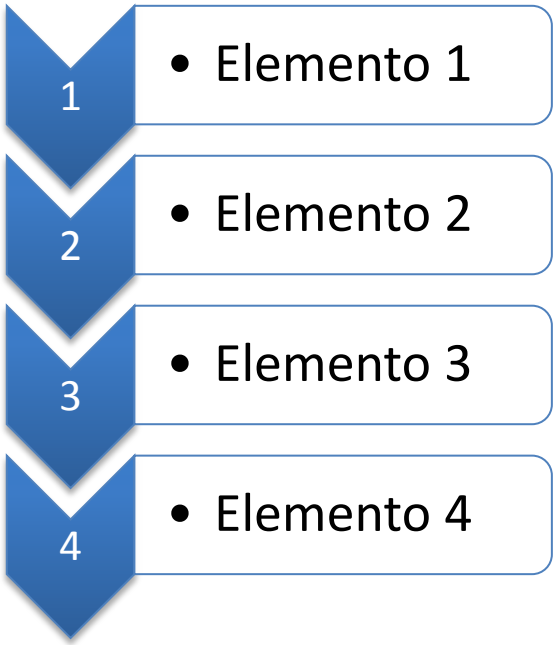
Listas

Sumário

- Conceito de listas
- Estruturas para armazenar
- Atividades
- Referências

Listas: conceito/características

- Uma lista é um conjunto de elementos:
  - Usualmente de um mesmo tipo
  - Possui uma ordem
    - Primeiro elemento
    - Último elemento
  - Elementos intermediários
    - Antecessor
    - Sucessor
- Estrutura Linear!



Listas lineares

- Uma lista linear é um conjunto de n elementos L[0], L[1], ..., L[n-1] tais que
  - n > 0 e L[0] é o primeiro elemento
  - para 0 < k < n, L[k] é precedido por L[k-1]

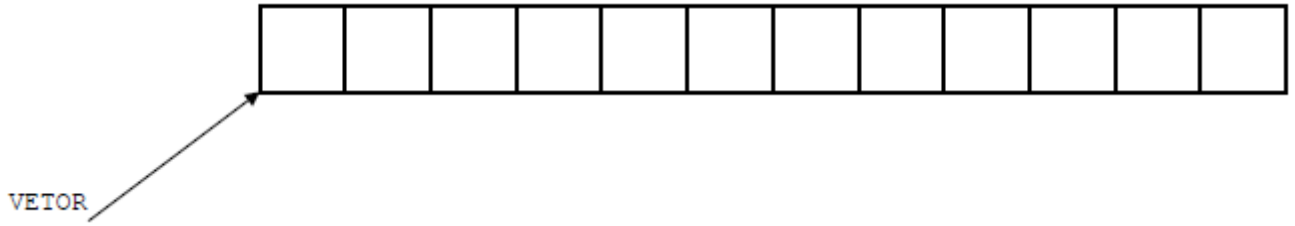
- Agrupam informações referentes a um conjunto de elementos que, de alguma forma, se relacionam.
  - Estáticas (alocação sequencial)
  - Dinâmicas (alocação encadeada)

Estrutura para armazenar

- Se sabemos o tamanho máximo da lista...
  - Podemos alocar todo o espaço
  - Espaço "contíguo" na memória: sequencial
  - Podemos usar um vetor!
- Exemplo: armazenar até 10 notas de alunos

Estrutura para armazenar

- Vetor
  - ocupa um **espaço contíguo de memória**
  - permite **acesso randômico** aos elementos, a partir do ponteiro para o início (vetor[i]).
  - deve ser dimensionado com um **número fixo de elementos**: e se precisarmos de mais ou de poucos elementos?



Estrutura para armazenar

- Por que podemos usar vetor para as notas?
  - Tamanho máximo da lista: 10
  - Dados todos do mesmo tipo: float
- A lista vai estar sempre cheia?
  - Se houver só 7 notas, quantas imprimir?
  - Mas como vamos saber que são 7?
  - Variável de controle de quantidade

float notas[10];

int quantidade;

Estrutura para armazenar

float notas[10];

int quantidade;

0	1	2	3	4	5	6	7	8	9
8.9	8.1	7.8	9.1	6.6	5.4	9.0			

quantidade=7;

Implementada em uma Lista Encadeada

- Digamos que as células da lista são do tipo celula:

```
typedef struct reg {
   char         conteudo;
   struct reg *prox;
} celula;
```

- Nossa lista terá uma célula-cabeça e uma variável global apontará a cabeça da lista:

```
celula *pi;
```

Implementada em uma Lista Encadeada

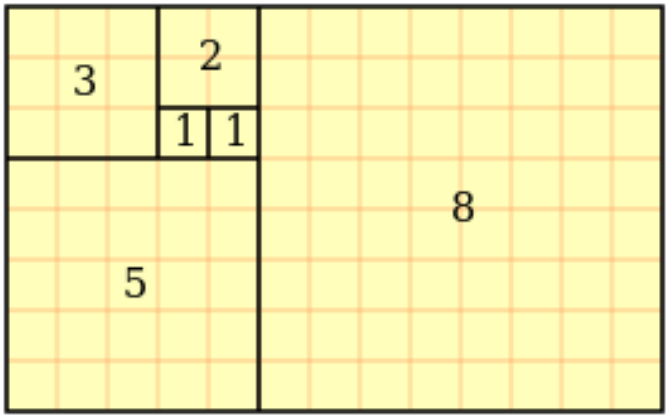
```
void criapilha (void) {
   pi = mallocc (sizeof (celula)); // cabeça
   pi->prox = NULL;
}

void empilha (char y) {
   celula *nova;
   nova = mallocc (sizeof (celula));
   nova->conteudo = y;
   nova->prox  = pi->prox;
   pi->prox = nova;
}

char desempilha (void) {
   char x;
   celula *p;
   p = pi->prox;
   x = p->conteudo;
   pi->prox = p->prox;
   free (p);
   return x;
}
```

Atividade!

- Implemente uma função recursiva que encontre o elemento apresentado na série de Fibonacci em uma dada posição.



- Série de Fibonacci:
  - F0 = F1 = 1
  - Fn = Fn-1 + Fn-2 para n > 1,
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

Referências

- CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José Lucas. **Introdução a Estrutura de Dados.** Editora Campus, 2004.
- MIZRAHI, V. V. **Treinamento em linguagem C.** São Paulo: Makron Books, 1990.