# Aplicações de Pilhas

## Estrutura de Dados — QXD0010

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará
Campus Quixadá

2º semestre/2025

# Balanceamento de parênteses e colchetes

## Definição Recursiva

Dizemos que uma sequência de parênteses e colchetes é uma sequência **bem-formada** se ela:

(i) é vazia

(ii) é uma sequência bem formada entre parênteses

(*sequência bem-formada*)

(iii) ou se é uma sequência bem formada entre colchetes

[*sequência bem-formada*]

# Balanceamento de parênteses e colchetes

Exemplos:

| Bem-formada | Malformada |
| --- | --- |
| | ( [ ] |
| [ [ ] ] | [ [ ) ) |
| ( [ ( ) ( [ ] ) ] ) | ( [ ) ] |

# Balanceamento de parênteses e colchetes

Exemplos:

| Bem-formada | Malformada |
| --- | --- |
| | ( [ ] |
| [ [ ] ] | [ [ ) ) |
| ( [ ( ) ( [ ] ) ] ) | ( [ ) ] |

Como usar pilha para testar se a sequência é bem-formada?

# Balanceamento de parênteses e colchetes

Exemplos:

| Bem-formada | Malformada |
|---|---|
| | ( [ ] |
| [ [ ] ] | [ [ ) ) |
| ( [ ( ) ( [ ] ) ] ) | ( [ ) ] |

Como usar pilha para testar se a sequência é bem-formada?

Para testar, leia cada símbolo e se:

# Balanceamento de parênteses e colchetes

Exemplos:

| Bem-formada | Malformada |
|---|---|
| | ( [ ] |
| [ [ ] ] | [ [ ) ) |
| ( [ ( ) ( [ ] ) ] ) | ( [ ) ] |

Como usar pilha para testar se a sequência é bem-formada?

Para testar, leia cada símbolo e se:

1. leu ( ou [: empilha o símbolo lido

# Balanceamento de parênteses e colchetes

Exemplos:

| Bem-formada | Malformada |
| --- | --- |
| | ( [ ] |
| [ [ ] ] | [ [ ) ) |
| ( [ ( ) ( [ ] ) ] ) | ( [ ) ] |

Como usar pilha para testar se a sequência é bem-formada?

Para testar, leia cada símbolo e se:

1. leu ( ou [: empilha o símbolo lido
2. leu ]: desempilha [

# Balanceamento de parênteses e colchetes

Exemplos:

| Bem-formada | Malformada |
|---|---|
| | ( [ ] |
| [ [ ] ] | [ [ ) ) |
| ( [ ( ) ( [ ] ) ] ) | ( [ ) ] |

Como usar pilha para testar se a sequência é bem-formada?

Para testar, leia cada símbolo e se:

1. leu ( ou [: empilha o símbolo lido
2. leu ]: desempilha [
3. leu ): desempilha (

# Implementação em C

# Implementação em C

```
1  bool bemFormada(char exp[]) {
2    CStack *p = cstack_create();
3    for(size_t i = 0; i < strlen(exp); i++) {
4      switch (exp[i]){
5          case ')': if(!cstack_empty(p) && cstack_top(p) == '('
6                        cstack_pop(p);
7                    else return false;
8                    break;
9          case ']': if(!cstack_empty(p) && cstack_top(p) == '['
10                       cstack_pop(p);
11                   else return false;
12                   break;
13         default : cstack_push(p, exp[i]);
14     }
15   }
16   bool is_empty = cstack_empty(p);
17   cstack_free(p);
18   return is_empty;
19 }
```

# Testando a Implementação

# Testando a Implementação

```
24 int main() {
25   char array[300];
26   scanf("%299s", array);
27   printf("bem formada? %s\n",
28     bemFormada(array) ? "true" : "false");
29   return 0;
30 }
```

# Notação de expressões

# Notação de expressões

# Notação de expressões

Exemplo 1:

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Prefixa: é a notação polonesa do lógico Jan Lukasiewicz

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Prefixa: é a notação polonesa do lógico Jan Lukasiewicz
   - Operador precede operandos

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Prefixa: é a notação polonesa do lógico Jan Lukasiewicz
   - Operador precede operandos

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Prefixa: é a notação polonesa do lógico Jan Lukasiewicz
   - Operador precede operandos
3. Posfixa: é notação polonesa reversa (RPN), das calculadoras HP.

# Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Prefixa: é a notação polonesa do lógico Jan Lukasiewicz
   - Operador precede operandos
3. Posfixa: é notação polonesa reversa (RPN), das calculadoras HP.
   - Operador sucede operandos

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

`2 * ( (2 + 1) * 4 + 1)`

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

`2 * ( 3 * 4 + 1)`

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

`2 * ( 12 + 1)`

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

`2 * 13`

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

$$26$$

Resolvendo com notação posfixa:

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação posfixa:

2

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação posfixa:

2 2

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

$$26$$

Resolvendo com notação posfixa:

$$2\ 2\ 1$$

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

$$26$$

Resolvendo com notação posfixa:

$$\texttt{2 2 1 +}$$

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação posfixa:

2 3

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação posfixa:

`2 3 4`

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

$$26$$

Resolvendo com notação posfixa:

`2 3 4 *`

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação posfixa:

2 12

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

$$26$$

Resolvendo com notação posfixa:

$$2 \ 12 \ 1$$

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

$$26$$

Resolvendo com notação posfixa:

$$\texttt{2 12 1 +}$$

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação posfixa:

2 13

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

`2 * ( (2 + 1) * 4 + 1)`

26

Resolvendo com notação posfixa:

`2 2 1 + 4 * 1 + *`

`2 13 *`

# Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

$$26$$

Resolvendo com notação posfixa:

$$26$$

# Calculando expressões posfixas

Algoritmo:

# Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:

# Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:
   - Se for número $n$:

# Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:
   - Se for número $n$:
     - empilha $n$

# Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:
   - Se for número $n$:
     - empilha $n$
   - Se for operador $\oplus$:

# Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:
   - Se for número $n$:
     - empilha $n$
   - Se for operador $\oplus$:
     - desempilha $operando_1$

# Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:
   - Se for número $n$:
     - empilha $n$
   - Se for operador $\oplus$:
     - desempilha $operando_1$
     - desempilha $operando_2$

# Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:
   - Se for número $n$:
     - empilha $n$
   - Se for operador $\oplus$:
     - desempilha $operando_1$
     - desempilha $operando_2$
     - empilha $operando_2 \oplus operando_1$

# Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:
   - Se for número $n$:
     - empilha $n$
   - Se for operador $\oplus$:
     - desempilha $operando_1$
     - desempilha $operando_2$
     - empilha $operando_2 \oplus operando_1$

2. Ao final, desempilha o único valor contido na pilha e retorna.

# Calculando expressões posfixas

Problema: Suponha dada uma expressão aritmética em notação posfixa sujeita às seguintes restrições:

1. cada número consiste nos inteiros do conjunto `0,1,...,9`;
2. os únicos operadores são $+$, $-$, $*$, $/$

Escreva uma função que calcule o valor da expressão.

# Solução — Implementação em C

# Solução — Implementação em C

```
1  // Supoe que 'posfix' contem expressao posfixa valida
2  double calculaPosfixa(char posfix[]) {
3      double a, b;
4      DStack *p = dstack_create();
5      for(size_t i = 0; i < strlen(posfix); i++) {
6          if( isdigit(posfix[i]) ) {
7              char c = posfix[i];
8              dstack_push( p, atof(&c) );
9          } else if(!isspace(posfix[i])) {
10             a = dstack_top(p); dstack_pop(p);
11             b = dstack_top(p); dstack_pop(p);
12             switch(posfix[i]) {
13                 case '+': dstack_push(p, b + a); break;
14                 case '-': dstack_push(p, b - a); break;
15                 case '*': dstack_push(p, b * a); break;
16                 case '/': dstack_push(p, b / a); break;
17             }
18         }
19     }
20     double d = dstack_top(p);
21     dstack_free(p);
22     return d;
23 }
```

# Converter uma expressão completamente parentizada para a notação posfixa

Objetivo:

`(1 + (((2 * 3) / 4) * 5))` $\Rightarrow$ `1 2 3 * 4 / 5 * +`

# Converter uma expressão completamente parentizada para a notação posfixa

Objetivo:

`(1 + (((2 * 3) / 4) * 5))` $\Rightarrow$ `1 2 3 * 4 / 5 * +`

Algoritmo:

- Copiamos os números diretamente na saída
- Quando aparecer '(' na entrada, ignoramos
- Quando aparecer um novo operador na entrada:
  - empilhamos o operador novo
- Quando aparecer ')' na entrada:
  - desempilhamos um operador, copiando para a saída

# Solução — Implementação em C

# Solução — Implementação em C

```
1  void ParentizadaParaPosfixa(char exp[], char saida[]) {
2      int index = 0;
3      CStack *p = cstack_create(); // guarda os operadores
4      for(size_t i = 0; i < strlen(exp); i++) {
5          switch(exp[i]) {
6              case '(': break;
7              case ')': saida[index] = cstack_top(p);
8                        index++;
9                        cstack_pop(p);
10                       break;
11             case '+':
12             case '-':
13             case '*':
14             case '/': cstack_push(p, exp[i]);
15                       break;
16             case ' ': break;
17             default : saida[index] = exp[i]; index++;
18         }
19     }
20     saida[index] = '\0';
21 }
```

# Lendo números com mais de 1 dígito

# Lendo números com mais de 1 dígito

```
1  double calculaPosfixa(char posfix[]) {
2      DStack *p = dstack_create();
3      for(size_t i = 0; i < strlen(posfix); i++) {
4          if( isdigit(posfix[i]) ) {
5              dstack_push(p, 0);
6              while( isdigit(posfix[i]) ) {
7                  char ch = posfix[i++];
8                  double t = 10*dstack_top(p);
9                  dstack_pop(p);
10                 dstack_push(p, t + atof(&ch) );
11             } i--;
12         } else if(!isspace(posfix[i])) {
13             double a = dstack_top(p); dstack_pop(p);
14             double b = dstack_top(p); dstack_pop(p);
15             switch(posfix[i]) {
16                 case '+': dstack_push(p, b + a); break;
17                 case '-': dstack_push(p, b - a); break;
18                 case '*': dstack_push(p, b * a); break;
19                 case '/': dstack_push(p, b / a); break;
20             }
21         }
22     }
23     double d = dstack_top(p); dstack_free(p); return d;
24 }
```

# Lendo números com mais de 1 dígito

# Lendo números com mais de 1 dígito

```
1  void ParentizadaParaPosfixa(char exp[], char saida[]) {
2      int index = 0;
3      CStack *p = cstack_create(); // guarda os operadores
4      for(size_t i = 0; i < strlen(exp); i++) {
5          switch(exp[i]) {
6              case '(': break;
7              case ')': saida[index++] = ' ';
8                        saida[index++] = cstack_top(p);
9                        cstack_pop(p);
10                       break;
11             case '+':
12             case '-':
13             case '*':
14             case '/': cstack_push(p, exp[i]);
15                       saida[index++] = ' ';
16                       break;
17             default : saida[index++] = exp[i];
18         }
19     }
20     saida[index] = '\0';
21 }
```

# Para casa: Convertendo de infixa para pós-fixa

Objetivo:

```
1 + 2 * 3 / 4 * 5 - 6  ⇒  1 2 3 * 4 / 5 * + 6 -
```

# Para casa: Convertendo de infixa para pós-fixa

Objetivo:

```
1 + 2 * 3 / 4 * 5 - 6  ⇒  1 2 3 * 4 / 5 * + 6 -
```

Ideia:

# Para casa: Convertendo de infixa para pós-fixa

Objetivo:

```
1 + 2 * 3 / 4 * 5 - 6  ⇒  1 2 3 * 4 / 5 * + 6 -
```

Ideia:

- Copiamos os números diretamente na saída

# Para casa: Convertendo de infixa para pós-fixa

Objetivo:

```
1 + 2 * 3 / 4 * 5 - 6  ⇒  1 2 3 * 4 / 5 * + 6 -
```

Ideia:

- Copiamos os números diretamente na saída
- Quando aparecer um operador na entrada:

# Para casa: Convertendo de infixa para pós-fixa

Objetivo:

```
1 + 2 * 3 / 4 * 5 - 6  ⇒  1 2 3 * 4 / 5 * + 6 -
```

Ideia:

- Copiamos os números diretamente na saída
- Quando aparecer um operador na entrada:
  - enquanto o operador no topo tiver precedência maior ou igual ao operador na entrada

# Para casa: Convertendo de infixa para pós-fixa

Objetivo:

```
1 + 2 * 3 / 4 * 5 - 6  ⇒  1 2 3 * 4 / 5 * + 6 -
```

Ideia:

- Copiamos os números diretamente na saída
- Quando aparecer um operador na entrada:
  - enquanto o operador no topo tiver precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída

# Para casa: Convertendo de infixa para pós-fixa

Objetivo:

```
1 + 2 * 3 / 4 * 5 - 6  ⇒  1 2 3 * 4 / 5 * + 6 -
```

Ideia:

- Copiamos os números diretamente na saída
- Quando aparecer um operador na entrada:
  - enquanto o operador no topo tiver precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída
  - empilhamos o operador novo

# Para casa: Convertendo de infixa para pós-fixa

Objetivo:

```
1 + 2 * 3 / 4 * 5 - 6  ⇒  1 2 3 * 4 / 5 * + 6 -
```

Ideia:

- Copiamos os números diretamente na saída
- Quando aparecer um operador na entrada:
  - enquanto o operador no topo tiver precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída
  - empilhamos o operador novo
- No final, desempilhamos todos os elementos da pilha, copiando para a saída

# Para casa: Convertendo de infixa para pós-fixa

Objetivo:

```
1 + 2 * 3 / 4 * 5 - 6  ⇒  1 2 3 * 4 / 5 * + 6 -
```

Ideia:

- Copiamos os números diretamente na saída
- Quando aparecer um operador na entrada:
  - enquanto o operador no topo tiver precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída
  - empilhamos o operador novo
- No final, desempilhamos todos os elementos da pilha, copiando para a saída

Pergunta:

# Para casa: Convertendo de infixa para pós-fixa

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` $\Rightarrow$ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída
- Quando aparecer um operador na entrada:
  - enquanto o operador no topo tiver precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída
  - empilhamos o operador novo
- No final, desempilhamos todos os elementos da pilha, copiando para a saída

Pergunta:

- Como generalizar para o caso em que a expressão tem parênteses?

# Pilhas e Recursão

# Pilhas e recursão

Pergunta: Qual a relação entre *pilhas* e *recursão*?

# Pilhas e recursão

Pergunta: Qual a relação entre *pilhas* e *recursão*?

```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```

# Pilhas e recursão

Pergunta: Qual a relação entre *pilhas* e *recursão*?

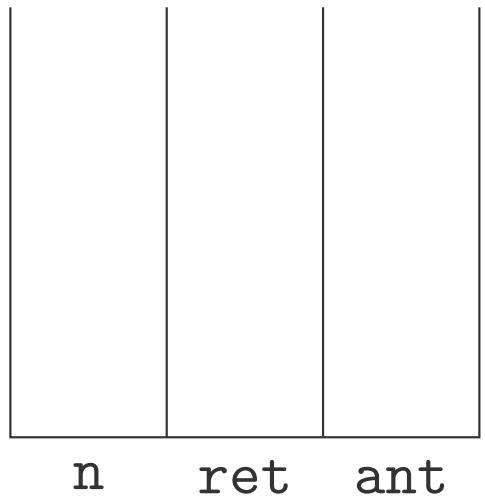
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```

Vamos tentar descobrir simulando uma chamada: `fat(4)`

# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

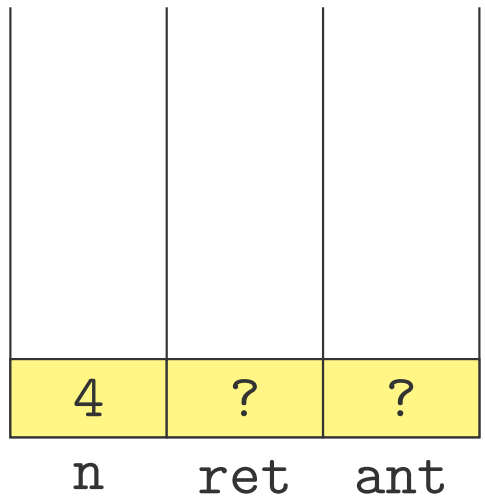
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

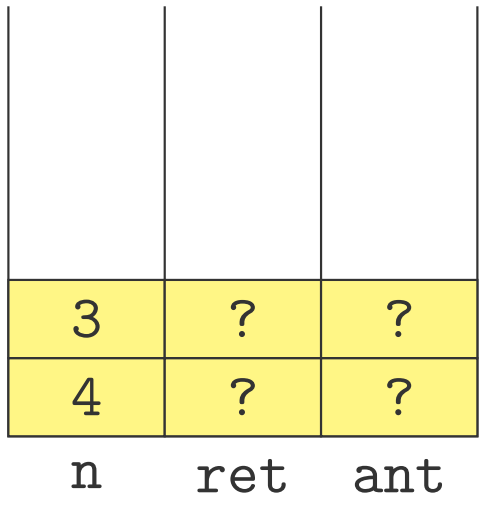
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

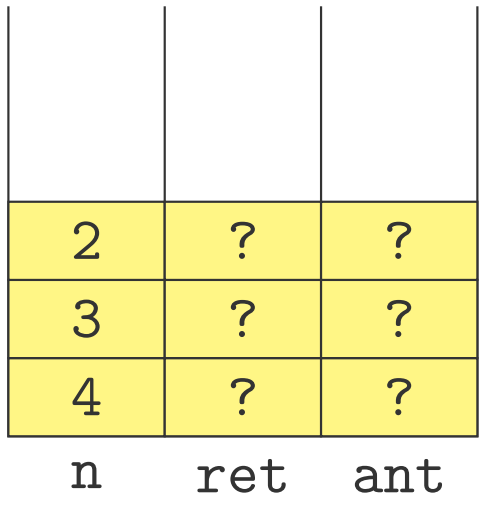
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

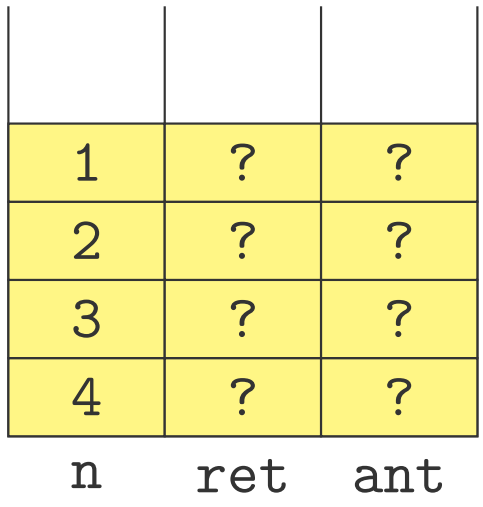
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

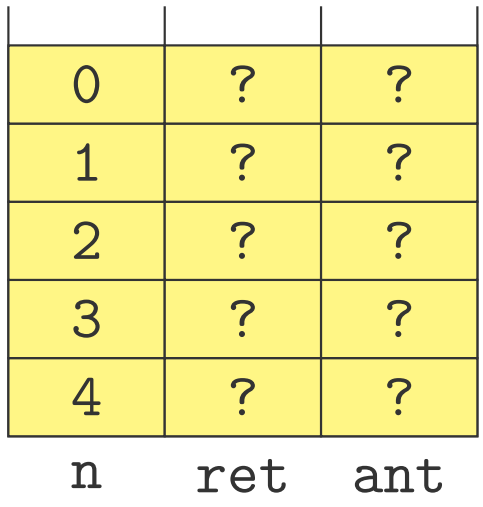
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

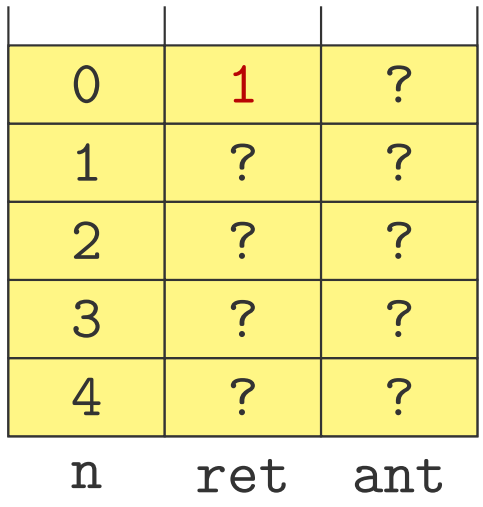
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```

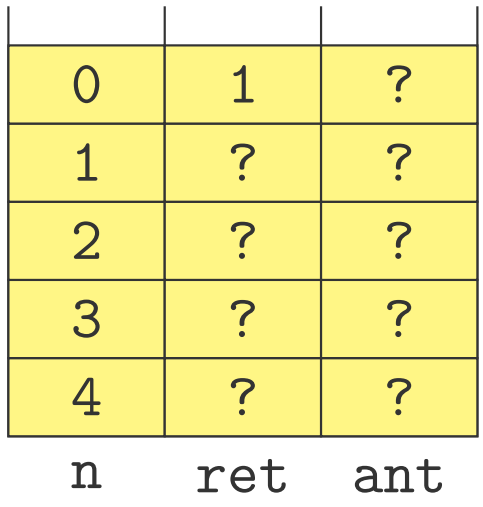


| n | ret | ant |
|---|---|---|
| 0 | 1 | ? |
| 1 | ? | ? |
| 2 | ? | ? |
| 3 | ? | ? |
| 4 | ? | ? |

# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

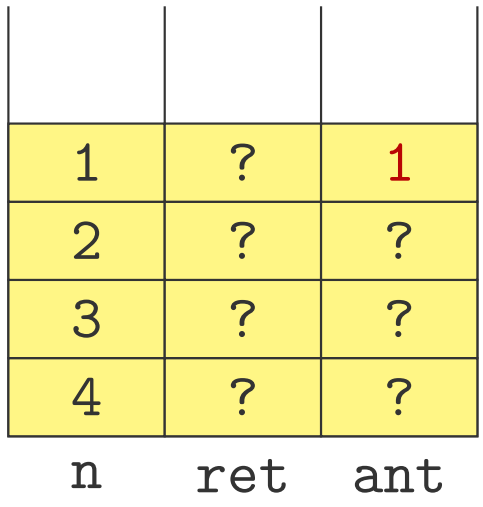
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

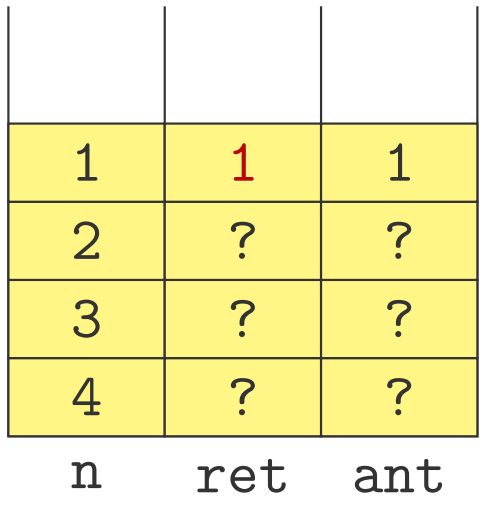
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

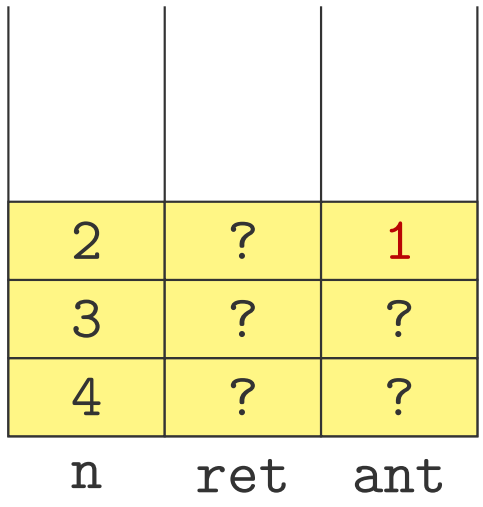
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

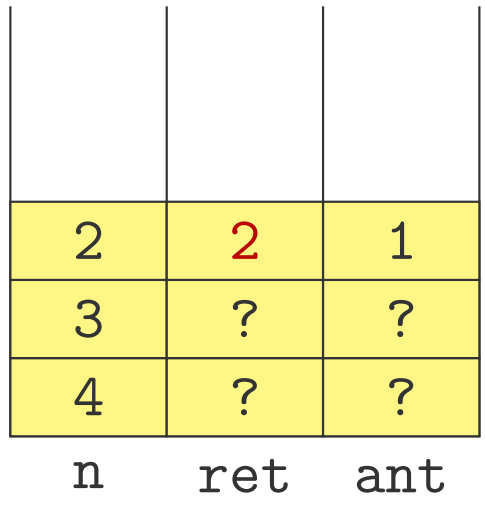
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

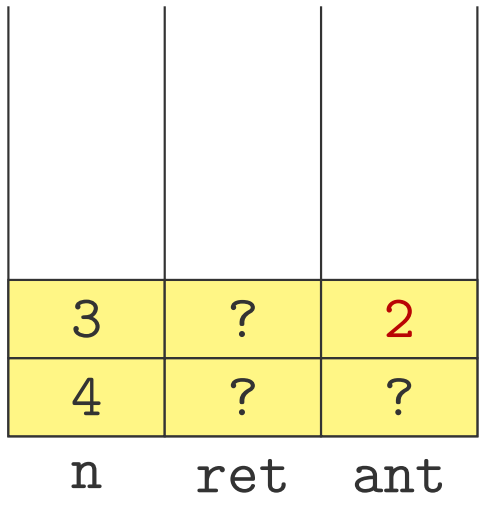
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

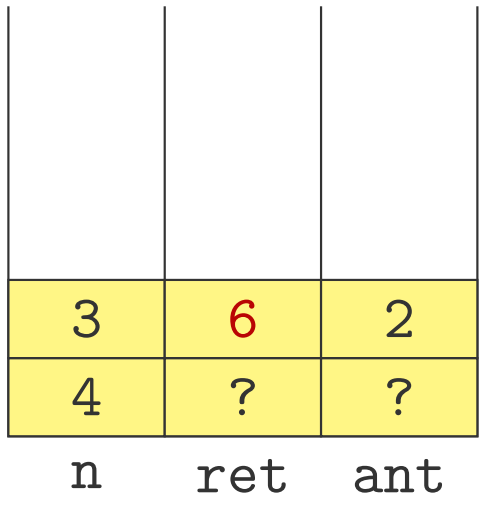
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

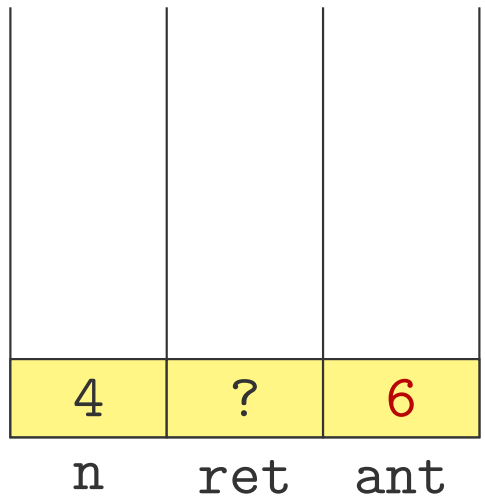
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

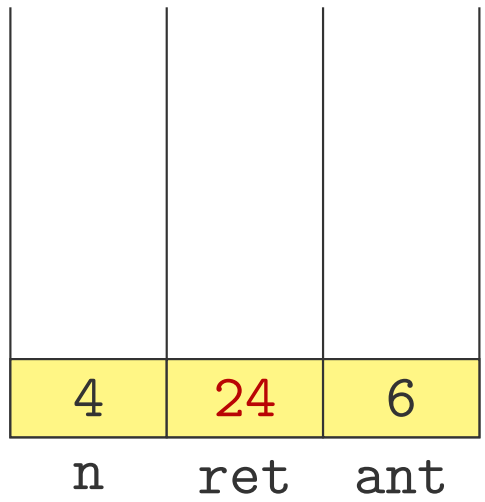
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



# Pilhas e recursão

Quando empilhamos:

# Pilhas e recursão

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

# Pilhas e recursão

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Quando desempilhamos:

# Pilhas e recursão

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Quando desempilhamos:

- Quando a chamada de `fat(n)` retorna, apagamos o espaço para as variáveis locais

# Pilhas e recursão

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Quando desempilhamos:

- Quando a chamada de `fat(n)` retorna, apagamos o espaço para as variáveis locais
- Restabelecemos os valores das variáveis locais para o valor que tinham antes da chamada

# Pilhas e recursão

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Quando desempilhamos:

- Quando a chamada de `fat(n)` retorna, apagamos o espaço para as variáveis locais
- Restabelecemos os valores das variáveis locais para o valor que tinham antes da chamada

O conjunto de variáveis locais formam um elemento da pilha

# Pilhas e recursão

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Quando desempilhamos:

- Quando a chamada de `fat(n)` retorna, apagamos o espaço para as variáveis locais
- Restabelecemos os valores das variáveis locais para o valor que tinham antes da chamada

O conjunto de variáveis locais formam um elemento da pilha

Isto é, a recursão pode ser simulada usando uma pilha de suas variáveis locais

# Exercícios

# Exercícios

(3) Implemente uma pilha **usando exatamente DUAS filas**. A pilha deve armazenar números inteiros. Não é necessário implementar todas as operações do TAD Pilha, as QUATRO únicas operações que são obrigatórias para esta questão são as operações `empilhar (push)`, `desempilhar (pop)`, `construir pilha (Construtor)` e `destruir Pilha (Destrutor)`.

**Observação:** Crie um arquivo **main.c** a fim de testar a estrutura de dados pilha que você criou.

**Atenção:** Não é para contruir a pilha suando listas encadeadas ou vetores. Pois nós já fizemos isso em aula. A única estrutura de dados permitida e que pode ser usada para resolver esse exercícios são as filas. Use EXATAMENTE duas filas. Nem mais nem menos que isso.

# Exercício

- Utilizando uma pilha, escreva uma função que verifique se uma string de entrada é da forma

$$str_1 C str_2$$

  tal que $str_1$ é uma string composta apenas por caracteres $A$ e $B$ e $str_2$ é a string reversa de $str_1$.

- Por exemplo, a cadeia $ABABBACABBABA$ é do formato especificado, enquanto as cadeias $ABABBACABB$, $ABA$, $BBBB$, $AAA$, $BBBBCAA$ e $ABBACBAABBBBBAB$ não seguem o formato.

- Sua função deve obedecer o seguinte protótipo:
  `bool str1Cstr2(char str[ ]);`

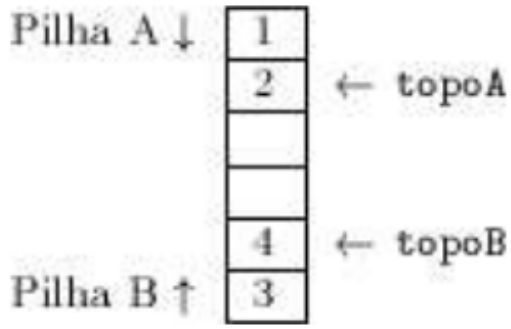
- Restrição: A string dada como entrada para a função deve ser percorrida uma única vez da esquerda para a direita.

# Exercício

- Faça um programa em C para ler um número inteiro maior que zero, converter este número de decimal para binário, usando pilha e apresentar na tela, o resultado da conversão.

# Exercício – Duas Pilhas em um vetor

Duas pilhas A e B podem compartilhar o mesmo vetor, como esquematizado na figura abaixo:




Faça as declarações de constantes e tipos necessárias e escreva as seguintes rotinas:

(a) `criaPilhas()`, que inicia os valores de topoA e topoB.

(b) `vaziaA()` e `vaziaB()`.

(c) `empilhaA(int x)` e `empilhaB(int x)`.

(d) `desempilhaA()` e `desempilhaB()`.

**Observação:** Só deve ser emitida uma mensagem de pilha cheia se todas as posições do vetor estiverem ocupadas.

# FIM