# Tipos Abstractos de Dados
# O TAD Stack

Sumário:
Revisões…
O conceito de tipo abstracto de dados.
O TAD Stack: Sua definição e  implementação com arrays. Exemplos de uso: conversão infix-postfix e  avaliação de expressões em postfix.

# Recursão

- A maioria das funções Matemáticas são definidas através duma fórmula. Por exemplo a conversão Celsius/Fahrenheit é dada por:

$$C = \frac{5 \times (F - 32)}{9}$$

- Se quiser criar uma função em C que converta C<->F, uma linha de código é suficiente para esta definição

- Há outra forma de definir funções (def. recursivas)

$$f(x) = \begin{cases} 0 & x = 0 \\ 2f(x-1) + x^2 & x > 0 \end{cases}$$

# Recursão

- Desta definição podemos concluir que:
  - $f(0) = 0$
  - $f(1) = 1$

$$f(x) = \begin{cases} 0 & x = 0 \\ 2f(x-1) + x^2 & x > 0 \end{cases}$$

  - $f(2) = 2f(1) + 2^2 = 6$
  - $f(3) = 2f(2) + 3^2 = 12 + 9 = 21$

- Uma função que se defina em termos de si própria, diz-se recursiva.

- O C tem mecanismos que permitem a implementação de funções recursivas (recursão mais genericamente) de forma transparente no uso

# Recursão

```
int f(int x){
    if(x==0)
        return 0;
    else
        return 2*f(x-1)+x*x;
}
```

- A cláusula if é chamado de caso de base (resolvido sem recurso à recursão)
- a cláusula else é o caso geral(e recursivo)
- O facto de definirmos uma função em termos de si própria, não é uma “pescadinha de rabo na boca”.
- Mas uma má definição recursiva poderá ser. Desde que o cálculo da função evolua para o caso de base, e as linguagens tenham mecanismos para implementar a recursão, é possível avaliar as funções recursivas.

# Recursão

- Só quando tenho definições do tipo f(5) em termos de f(5), é que tenho asneira (experimente modificar o código dado para definir f(n) em termos de f(n) e veja o que acontece…

Esgotamos a memória ….

- Mas se definirmos f(n) em função de f(k), k<n, conseguirmos atingir o caso de base, as sucessivas chamadas da função que ficam suspensas, para o cálculo das chamadas de ordem inferior poderem ser calculadas.

- No caso da função f, calcular f(5), significa que tem de ser calculado 1º f(4), para avaliar f(5).

# Recursão

- Para avaliar f(4) é necessário avaliar f(3), f(2), f(1) e finalmente f(0).

- f(0) é o caso de base, e é calculado. Retornado f(0) pode ser calculado f(1), após retornar f(1) é calculado f(2), depois f(3), depois f(4) e finalmente f(5)

- Após esta descrição não é difícil focarmo-nos na eficiência da recursão, mas

- A seu tempo falaremos de eficiência..

# Recursão

- Qual a avaliação de f(-2)?
  - Para avaliar f(-2), chamo f(-3)
    - Para avaliar  f(-3) chamo f(-4). . .

- Na definição da função também não sabemos calcular f(-2). -2 não pertence ao domínio da função…

- Há que garantir que as sucessivas chamadas recursivas evoluem para o caso de base.

- Caso contrário temos infinitas(??) chamado recursivas que “estoiram” com a memória.

# Recursão

- Pode acontecer que a definição das funções recursivas seja subtilmente errada. Exemplo:

```
int bad(int n){
    if(n==0)
        return 0;
    else
        return bad(n/3+1)+n-1;
}
```

- bad (0) é 0
- bad(1) é bad(1/3+1)+1-1=bad(1)
  - isto sim é uma “pescadinha de rabo na boca”
- Mas o computador não sabe avaliar esta incongruência
- Uma vez mais vamos estoirar com a memória.

# Recursão

- As duas regras fundamentais da recursão
  - Caso(s) de Base: Devem existir sempre casos de base, que são calculados de forma explícita (i.e. sem recursão)
  - Progredir / Evoluir: Os casos recursivos devem evoluir para os casos de base, para garantir que o programa termina, e o cálculo é possível

# Recursão

- A utilização de definições recursivas, não tem grande dificuldade

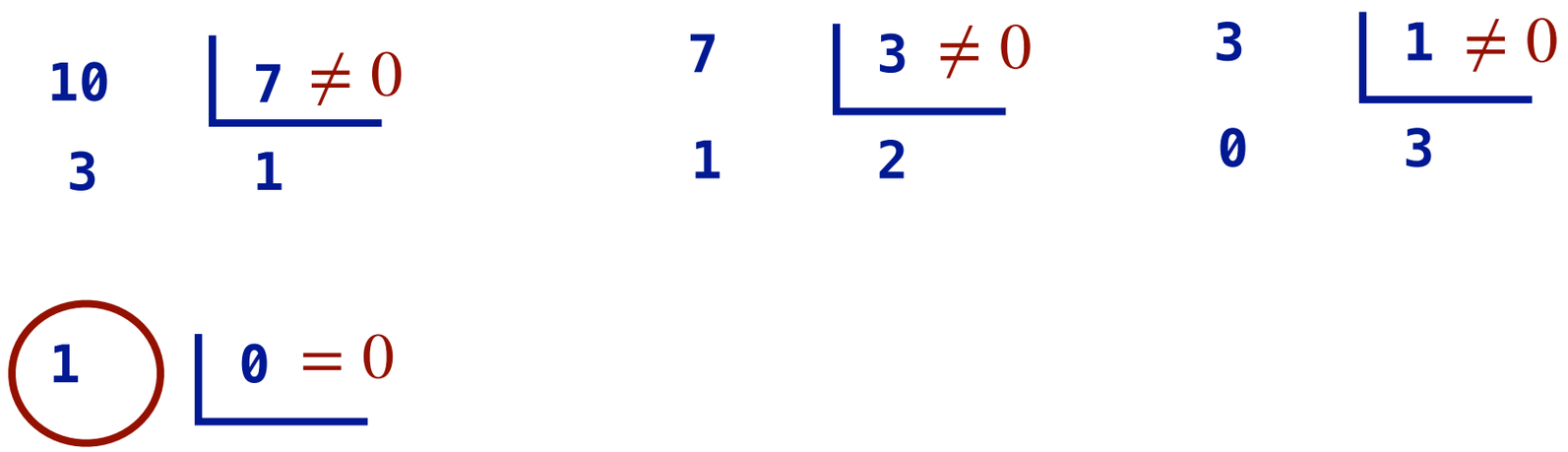
- A dificuldade existe em construir soluções recursivas para os problemas, porque estamos a dar a solução com o próprio problema

- Uma vez mais é o evoluir/progredir que funciona como modo de resolver o problema.

- Iremos usar durante o semestre variadas vezes a recursão, sobretudo se as estruturas são em si recursivas…

# Algoritmo de Euclides

- mdc(10,7)=**1**




```
int mdc(int a, int b){
     if(b==0)
          return a;
     else
          return mdc(b,a%b);
}
```

# Tipos Abstractos de Dados

- O que é um TAD?
  - TAD = **Conjunto de Operações**
  - Abstracções Matemáticas
    - Exemplo Conjuntos(Set)
      - operações?
- Para que servem?
  - Permitem a programação modular
  - para serem usados por outros módulos que deles necessitem

# Uso dum Tipos Abstractos de Dados

- Na prática requer uma implementação(O TAD tem implementação! definição

- Em si não especifica o modo como as operações são implementadas

- Não existe nenhuma regra que nos diga quais as operações suportadas por um TAD, trata-se duma opção de desenho.

- Na prática, serão necessários alguns passos para definir  o TAD, mas a ideia é definir os protótipos das operações…

- Depois implementar, para usar…

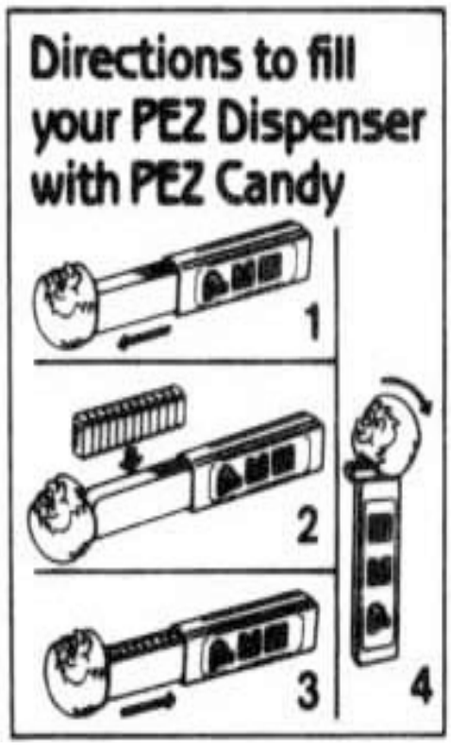
# O TAD Stack (pilha)

- O que é uma pilha?
  - Já vimos e lidámos com muitas:
    - Pilha de roupa

# O Tad Stack

- pilha de pratos

- distribuidor de pez


# O TAD Stack

- Uma stack, é uma estrutura de dados, que usa o protocolo "LAST IN FIRST OUT" para aceder aos seus elementos

  - o que significa este protocolo, com

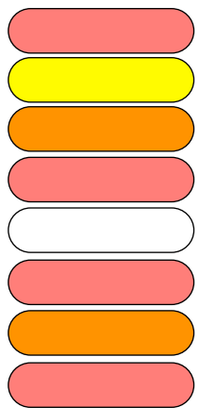
  ou

Os últimos são os primeiros!

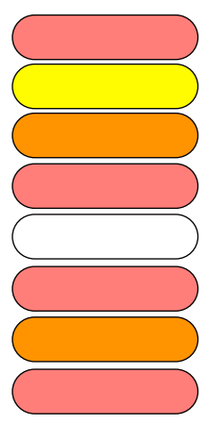
# O TAD Stack (pilha)

- Que operações fundamentais com uma pilha?
  - Examinar/consular o último elemento inserido
  - Retirar o último objecto inserido
  - Adicionar mais um elemento à stack



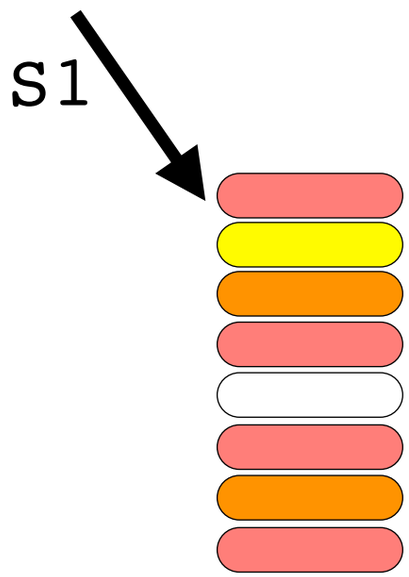
# O TAD Stack (pilha)

- Dar nomes às operações:
  - Ver o último (TOP)
  - Retirar o último (POP)
  - Inserir um element (PUSH)



# O TAD Stack (pilha)

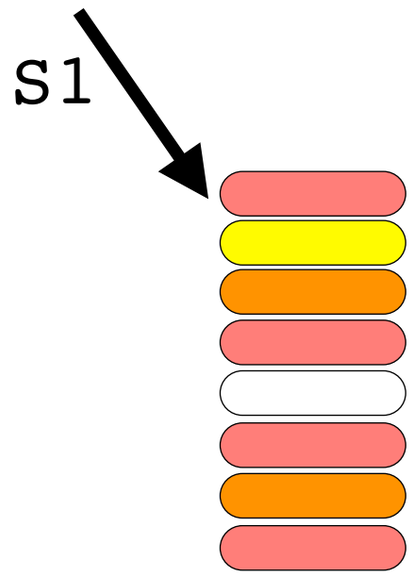
- Qual o resultado de fazer **Top** sobre a stack da figura?



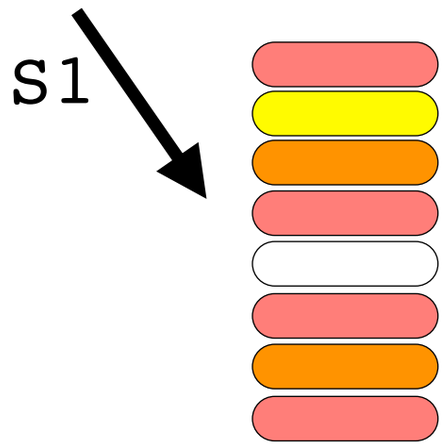

**Top(S1)** obtenho 

**Trata-se duma operação que não modifica a pilha**

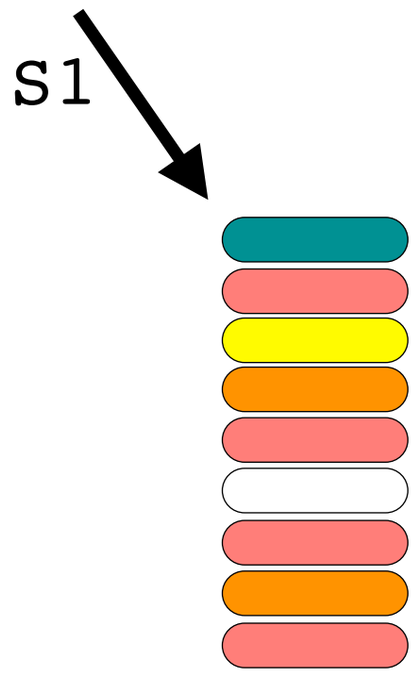
após **Top(S1)**



# O TAD Stack (pilha)



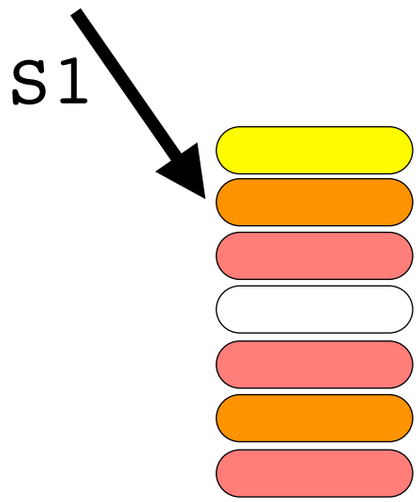

**Push (** **,S1)** resulta




**Top(S1)** dá

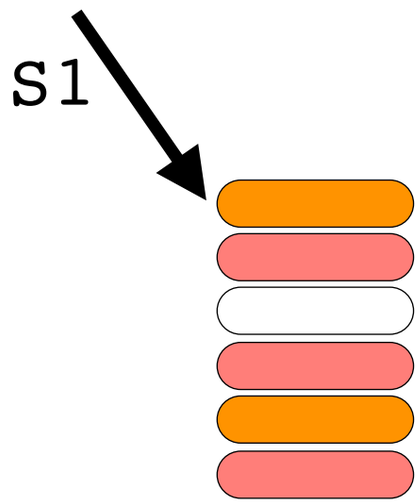
# O TAD Stack (pilha)

- Também a operação de retirar um elemento da pilha a modifica:




**Pop(S1)** resulta

**A pilha é modificada**




- Agora:

**Top(S1)** dá

# Implementação de Stacks

- A implementação duma stack usando arrays é trivial, mas:
  - Associado a cada stack estará o seu topo(TOS)
  - A criação duma nova pilha deve resultar um objecto deste tipo que não contenha elementos (que esteja vazia!, TOS=-1)
- Uma operação que informe se a pilha está vazia é usual e ajuda a controlar a estrutura (p.e. IsEmpty)
- Algumas operações nem podem ser realizadas se a estrutura estiver vazia (quais?)
- Também a operação de Push, quando realizada sobre um array que esteja cheio, fará crashar o programa

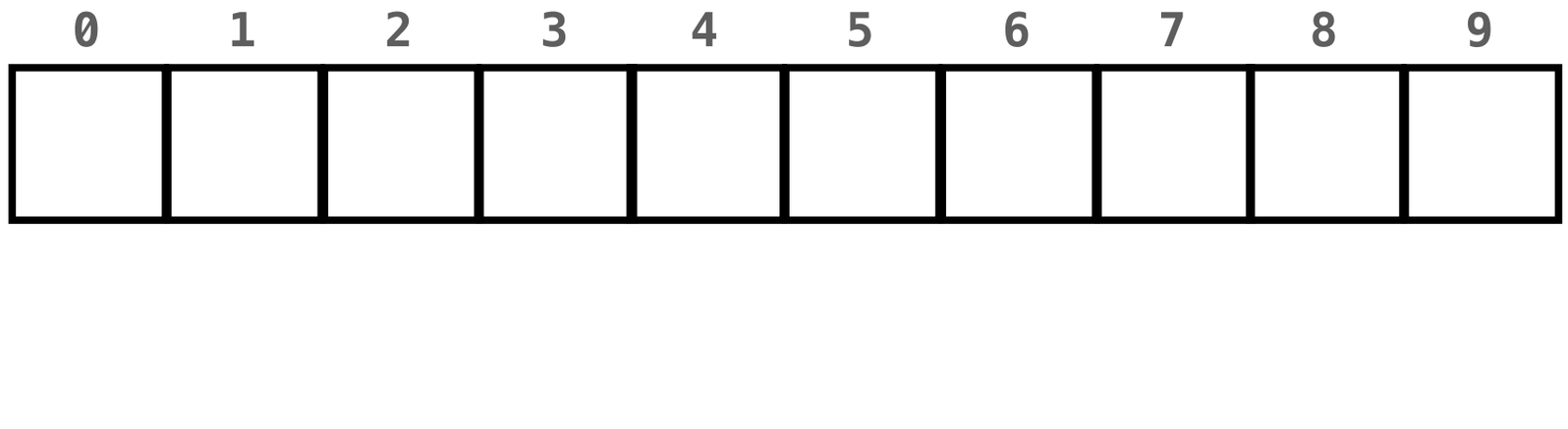
# Uma definição para Stack

```
typedef int ElementType;

#ifndef _Stack_h
#define _Stack_h

struct StackRecord;
typedef struct StackRecord *Stack;

int IsEmpty( Stack S );
int IsFull( Stack S );
Stack CreateStack( int MaxElements );
void DisposeStack( Stack S );
void MakeEmpty( Stack S );
void Push( ElementType X, Stack S );
ElementType Top( Stack S );
ElementType Pop( Stack S );

#endif
```

# Implementação com array

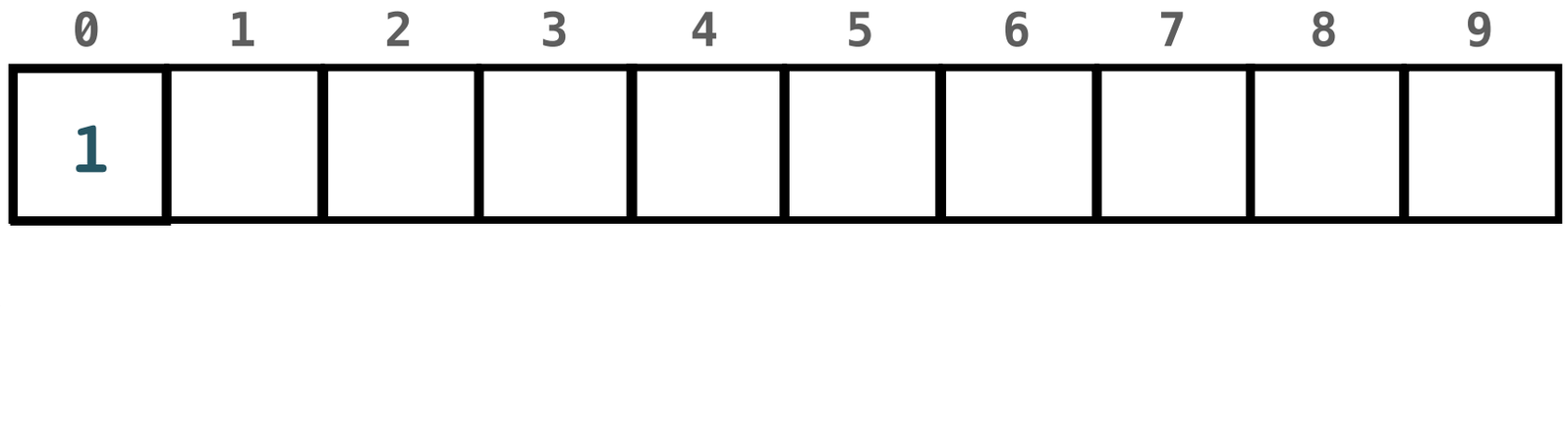
- Uma Stack vazia de tamanho 10:

```
Stack S1=CreateStack(10);
```




- Adicionar o inteiro 1 à stack

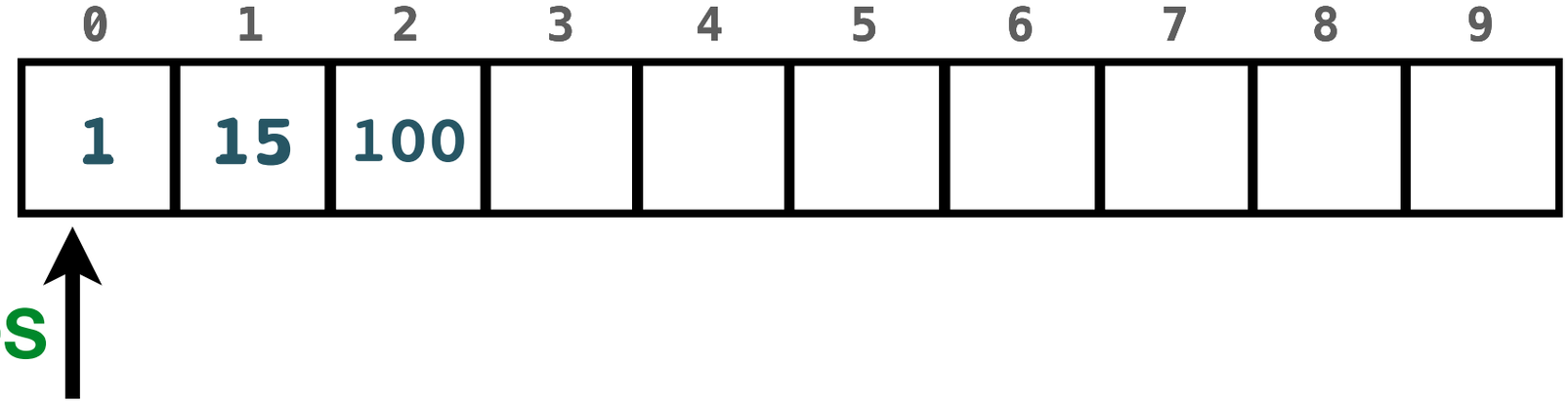
```
Push(1,S1);
```



# Implementação com array

- Adicionar mais:

```
Push(15,S1);

Push(100,S1);
```




- 

```
printf("Top of stack is %d \n",Top(S1));
```

output:

```
Top of stack is 100
```

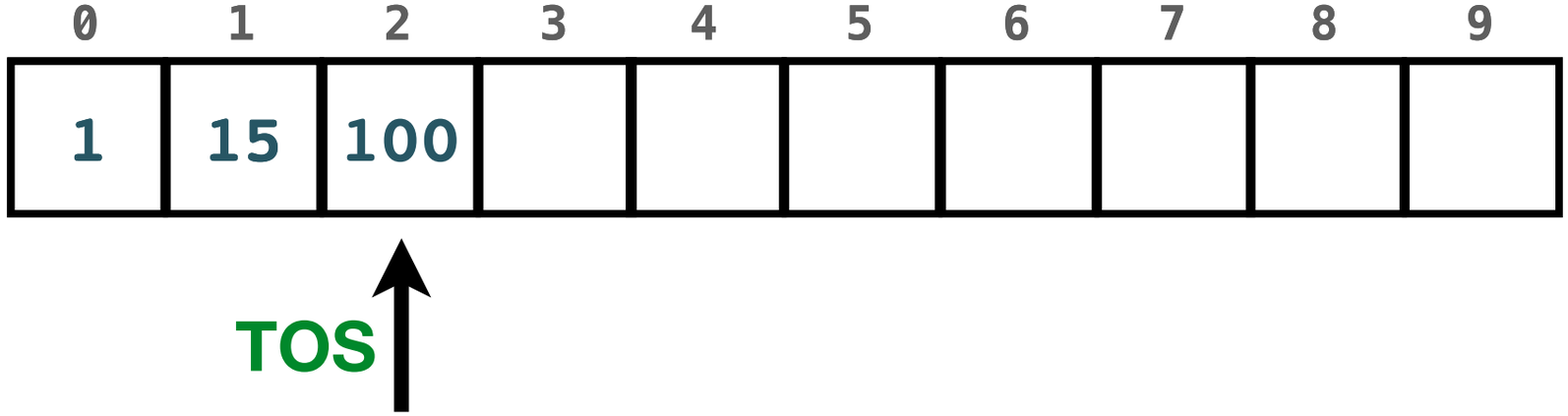
# Implementação com array

- Remover o último elemento inserido:

```
printf("Faz pop  %d \n",Pop(S1));
```

output:

```
Faz pop  100
```



# Aplicações das Stacks

- É frequente os compiladores identificarem que os nossos códigos faltam ( } ou comentários abertos que não foram fechados…

- Se tiverem um bom editor, nem é preciso compilar para sabermos que há algo de errado com o balanço dos parêntesis e afins (/*)

- Se quisermos verificar se um texto está correctamente balanceado, o uso duma stack é uma forma simples para fazer tal verificação.

- Assuma-se que queremos verificar o balanço entre {}, () e [] num qualquer texto(deixemos os comentários de fora…)

# Aplicações das Stacks

- A expressão “([])” mesmo não tendo qualquer significado está correctamente balanceada

- “)(“ não está

- por isso não vale a pena contar o nº de parêntesis a abrir e o nº de parêntesis a fechar para verificar o balanço

- Salvo manipulação de Strings(array de char) e funções que validem se se trata de parêntesis abrir ou a fechar, e o que fecha com o quê, temos o seguinte pseudo-código para verificar o balanço

# Aplicações das Stacks

```
//pseudo-código
Stack S1;
char str[];

  foreach (ch in str){
     if(isOpenParen(ch))
       Push(ch,S1);

     else{
       if(isCloseParen(ch)){
          if(!IsEmpty(S1){
              t=Pop(S1);
              if(match(ch,t))
                //good
               else
                //mau maria;
          }
          else
            //mau maria;
     }
 }//foreach

if(!IsEmpty(S2))
    //mau maria
```

# Aplicações das Stacks

- Conversão infix-postfix e avaliação de expressões:
  - A expressão (3+4)*10 diz-se em notação infix, por os operadores(binários)  estarem no meio dos operandos. Este tipo de notação, útil para nós, torna o cálculo de expressões complexo, por exigir o uso de parêntesis para estabelecer a ordem das operações.
  - 3 4 + 10 *  é o equivalente à expressão anterior em notação postfix

# Expressões postfix

- A avaliação duma expressão postfix pode fazer-se usando uma stack e com uma só passagem pelo input

- Faz-se, lendo a expressão: se é lido um número põe-se na Stack se é lido um operador, realiza-se a operação com os dois operandos retirados da Stack, e adiciona-se à Stack o resultado da operação

```
if (input é operando)
    Push(operando,S)
else //é operador
    op2=Pop(S)
    op1=Pop(S)
    resultado=op1 operador op2
    Push(resultado,S)
```

# Exemplo

- Exemplo:

- avaliar a expressão : 6 5 3 2 + 9 * 3 / - *

6

# Exemplo

- Exemplo:

- avaliar a expressão : 6 5 3 2 + 9 * 3 / - *

# Exemplo

- Exemplo:

- avaliar a expressão : 6 5 3 2 + 9 * 3 / - *

| 3 |
|---|
| 5 |
| 6 |

# Exemplo

- Exemplo:

- avaliar a expressão : 6 5 3 2 + 9 * 3 / - *

| 2 |
|---|
| 3 |
| 5 |
| 6 |

# Exemplo

- Exemplo:

- avaliar a expressão : 6 5 3 2 + 9 * 3 / - *

| |
|---|
| 2 |
| 3 |
| 5 |
| 6 |

+ = 5

# Exemplo

- Exemplo:

- avaliar a expressão : 6 5 3 2 + 9 * 3 / - *

| 9 |
|---|
| 5 |
| 5 |
| 6 |

# Exemplo

- Exemplo:

- avaliar a expressão : 6 5 3 2 + 9 * 3 / - *

| 9 |
|---|
| 5 |
| 5 |
| 6 |

*

=

| 45 |
|---|

# Exemplo

- Exemplo:

- avaliar a expressão : 6 5 3 2 + 9 * 3 / - *

| 3 |
|---|
| 45 |
| 5 |
| 6 |

# Exemplo

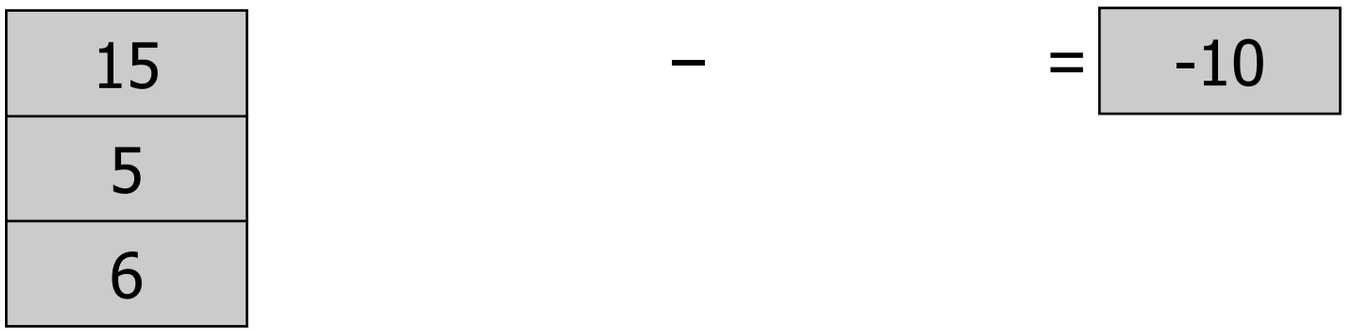
- Exemplo:

- avaliar a expressão : 6 5 3 2 + 9 * 3 / - *

# Exemplo

- Exemplo:

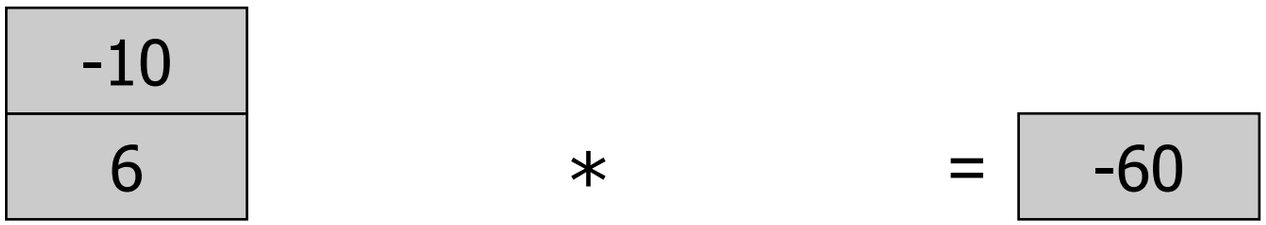
- avaliar a expressão : 6 5 3 2 + 9 * 3 / - *



# Exemplo

- Exemplo:

- avaliar a expressão : 6 5 3 2 + 9 * 3 / - *



# Exemplo

- Exemplo:

- avaliar a expressão : 6 5 3 2 + 9 * 3 / - *

-60

**A Stack contem um único valor que corresponde à avaliação da expressão**

**Mais do que um valor na stack significa que existe um erro.**

# Conversão infix-postix

- A expressão 6 5 3 2 + 9 * 3 / - * é a conversão para postfix da expressão em infix 6*(5-(3+2)*9/3)

- A conversão  duma expressão infix para postfix faz-se também usando uma stack cujo uso está descrito no algoritmo. Assume-se que só existem parênteses curvos e que a expressão está correcta (pelo menos correctamente balanceada de parênteses).

# Considerações

- input: sequência símbolos, i.e. números, operadores e parêntesis, que constituem a expressão em infix, que queremos converter para postfix

- output: sequência de números e operadores correspondente à tradução para postfix da sequência de input.

- A stack usada irá conter somente operadores e parentesis (abrir!)

# CONVERSÃO INFIX-POSTFIX

Prioridades:

(

* /

\+ –

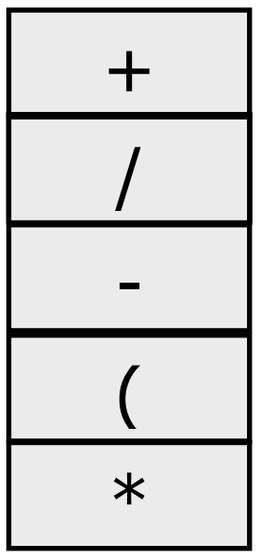
```
// Stack ST

while ((s=símbolo lido)!=null)
    if (s é numero) then
        output+=s
    else
        if (s == ‘)’) then
            while (Top(ST)!= ‘(‘)
                output += Pop(ST)
            Pop(ST)
        else
            if (prioridade(s) >prioridade(Top(ST)))
                Push(s,ST)
            else
                while(Top(ST)<>”(“ or
                        prioridade(Top(ST))>=prioridade(s) )
                    output+=Pop(ST)
                Push(s,ST)

while (!IsEmpty(ST))
   output+=Pop(ST)
```

input:

| 6 | * | ( | 15 | - | ( | 3 | + | 12 | ) | * | 9 | / | 3 | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

output:

| 6 | 15 | 3 | 12 | + | 9 | * | 3 | / | - | * |
|---|---|---|---|---|---|---|---|---|---|---|



# Exercício

- Converter para postfix e avaliar
  - (10-(5+3)*4)/(13-(2*(3-4)))
  - postfix
    - 10 5 3 + 4 * - 13 2 3 4 - * - /
- avaliação a expressão em postfix
  - -22/15=-1.4666667

# Aplicações das Stacks

- Stacks de execução:

- Trata-se duma Stack onde são mantidas informações relativamente às variáveis locais (funções) e outras informações necessárias. Durante a execução dum programa é mantida uma stack cujos elementos são descritores das funções invocadas e activos, estes descritores são designados por frames.

# Aplicações das Stacks

- Recursão:

```
unsigned long fact(unsigned int n){
    if (n==0)
        return 1;
    else
        return n*fact(n-1);

}
```

  - A stack permite que o método fact  exista em diferentes frames activas. Cada frame armazena o parâmetro n e o valor a ser retornado(para simplificar!)

- Desde que as sucessivas chamadas evoluam para o caso de base, não há problema.

# Aplicações das Stacks

```
unsigned long fact(unsigned int n){
    if (n==0)
        return 1;
    else
        return n*fact(n-1);

}
```

```
int main() {
    // insert code here...
       printf("%ld\n", fact(5));

}
```

output:

```
120
Program ended with exit code: 0
```

| n | 1 |
|---|---|
| RV | 1 |
| n | 2 |
| RV | 2*1 |
| fact (1) | 1 |
| n | 3 |
| RV | 3*2 |
| fact (2) | 2 |
| n | 4 |
| RV | 4*6 |
| fact (3) | 6 |
| n | 5 |
| RV | 5*24 |
| fact (4) | 24 |