AULA 9

Pilhas

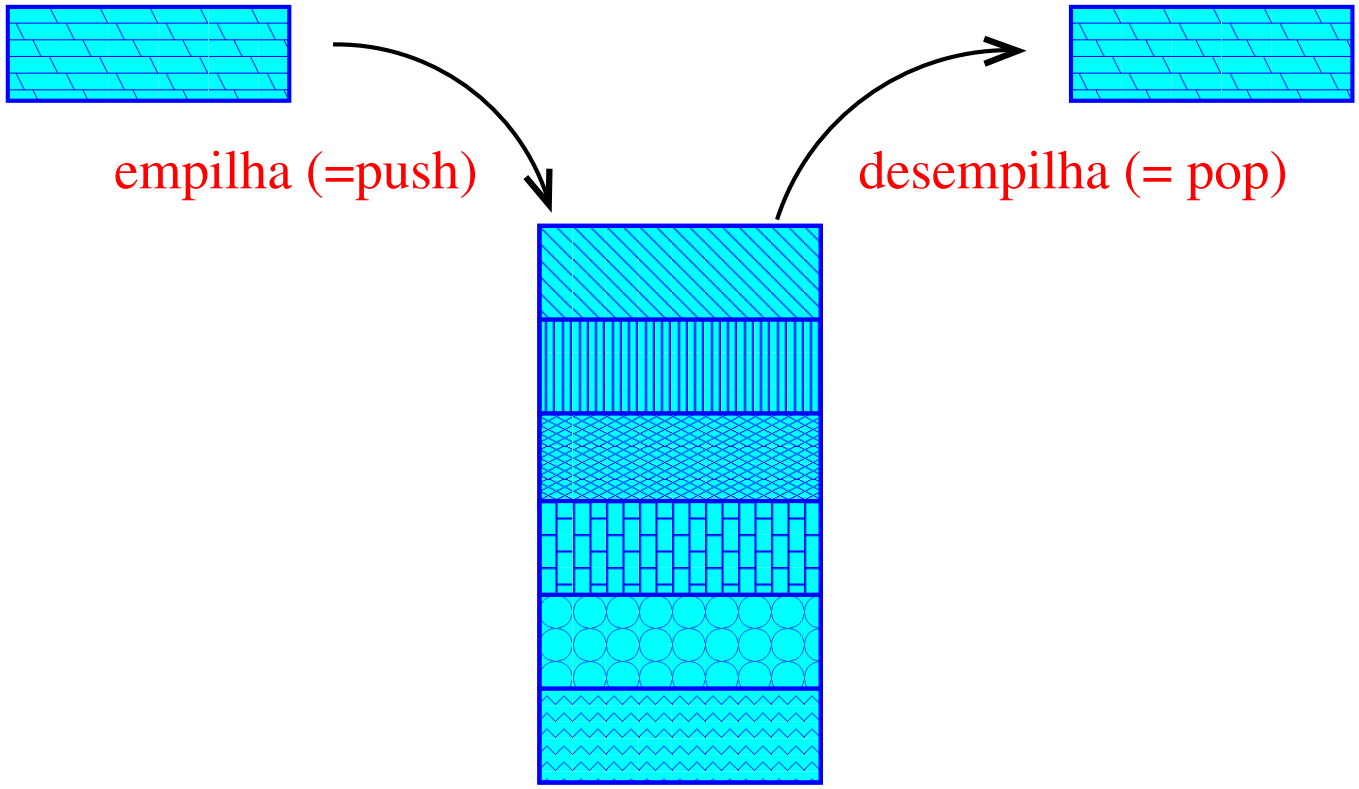
Fonte: http://dontmesswithtaxes.typepad.com/

PF 6.1 e 6.3

http://www.ime.usp.br/~pf/algoritmos/aulas/pilha.html

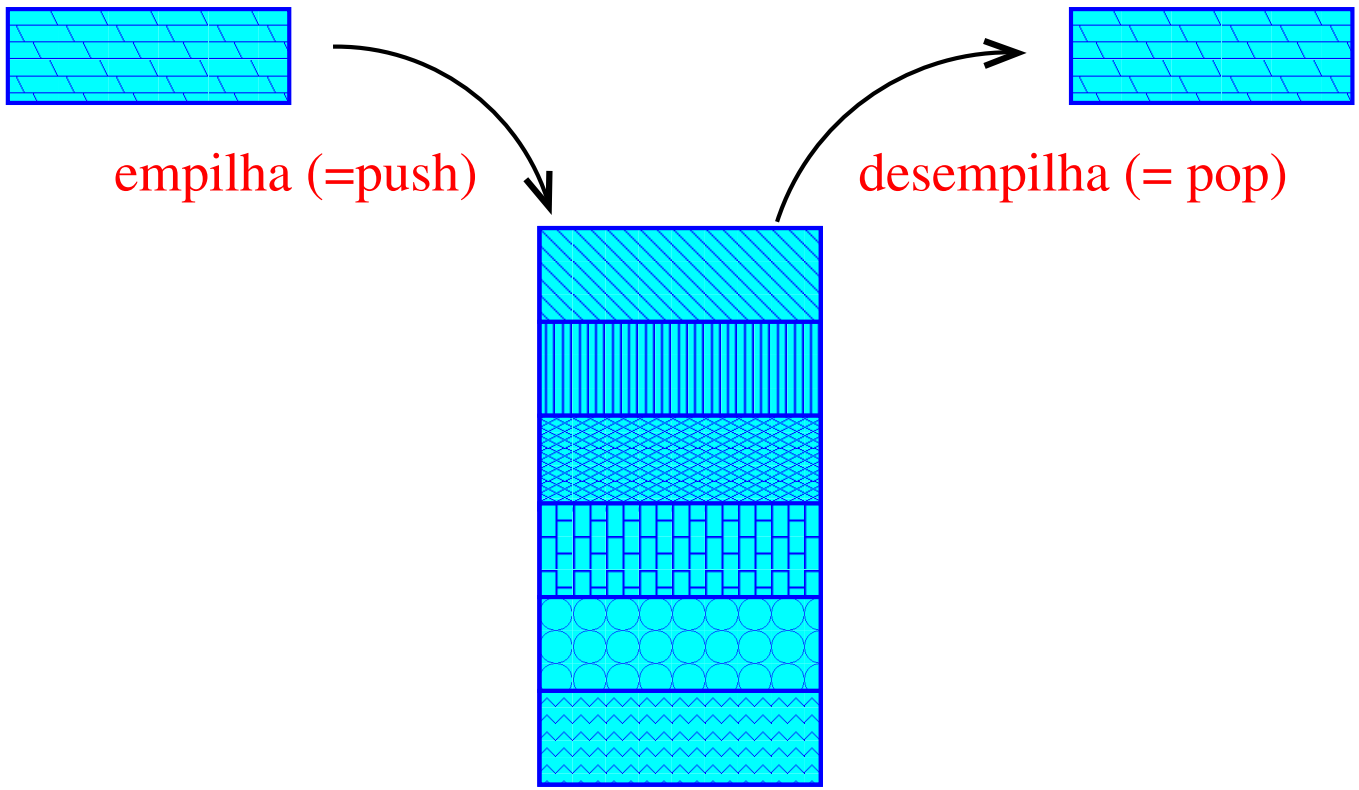
Pilhas

Uma **pilha** (=*stack*) é uma lista (=sequência) dinâmica em que todas as operações (inserções, remoções e consultas) são feitas em uma mesma extremidade chamada de **topo**.



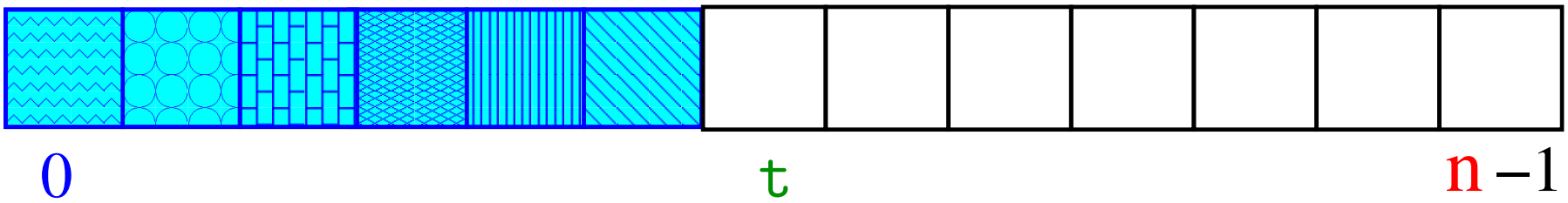
Pilhas

Assim, o primeiro objeto a ser removido de uma pilha é o último que foi inserido. Esta política de manipulação é conhecida pela sigla **LIFO** (=*Last In First Out*)



Implementação em um vetor

A pilha será armazenada em um vetor `s[0..n-1]`.



O índice `t` indica o topo (=*top*) da pilha.

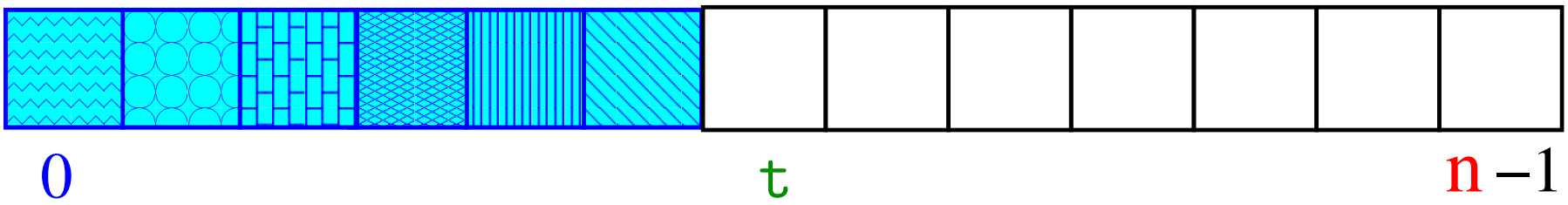
Esta é a primeira posição vaga da pilha.

A pilha está vazia se "`t == 0`".

A pilha está cheia se "`t == n`".

Implementação em um vetor

A pilha será armazenada em um vetor `s[0 . . n-1]`.



Para remover (=desempilhar=*pop*) um elemento faça

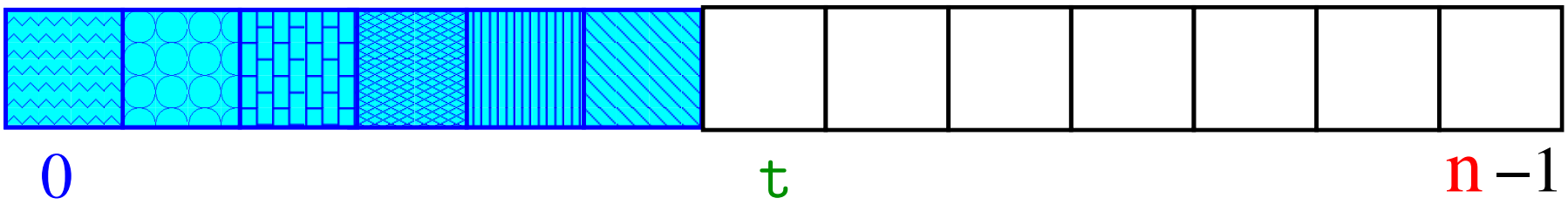
```
x = s[--t];
```

que é equivalente a

```
t -= 1;
x = s[t];
```

Implementação em um vetor

A pilha será armazenada em um vetor `s[0..n-1]`.



Para inserir (=empilhar=*push*) um elemento faça

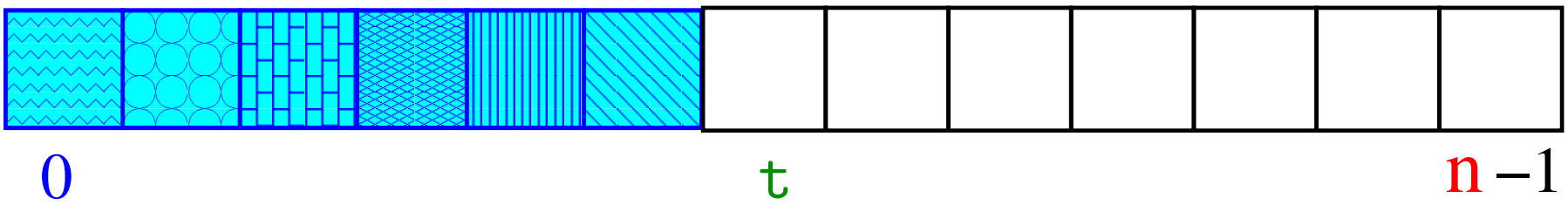
```
s[t++] = x;
```

que é equivalente a

```
s[t] = x;
t += 1;
```

Implementação em um vetor

A pilha será armazenada em um vetor `s[0..n-1]`.

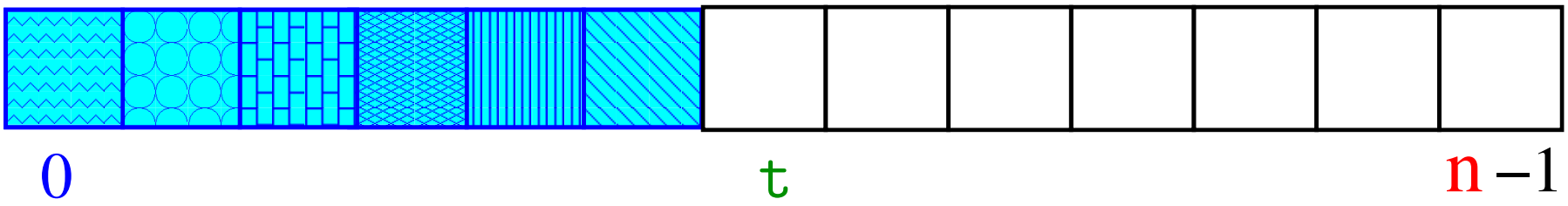


Para consultar um elemento, sem removê-lo, faça

```
x = s[t-1];
```

Implementação em um vetor

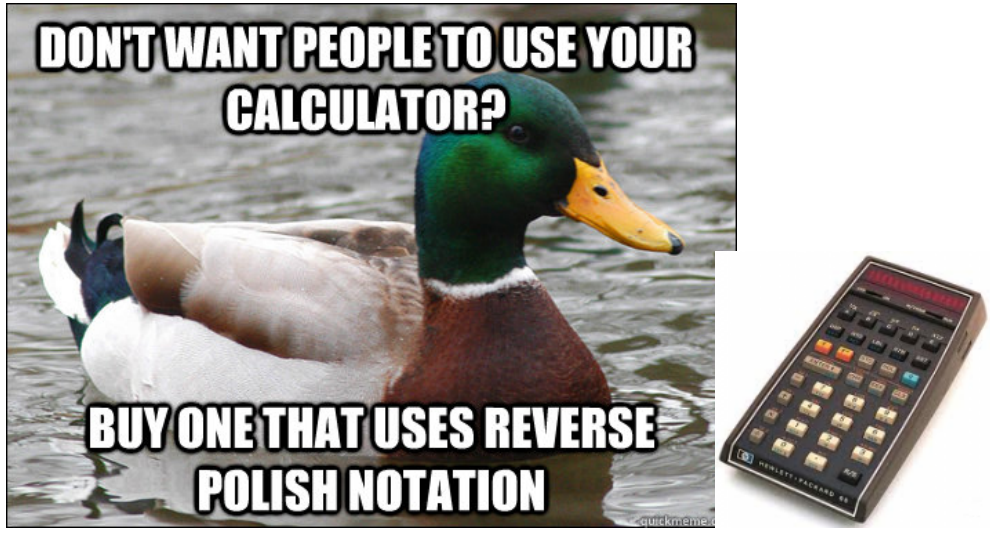
A pilha será armazenada em um vetor `s[0 . . n-1]`.



Tentar desempilhar de uma pilha que está vazia
é um erro chamado *stack underflow.*

Tentar empilhar em uma pilha cheia
é um erro chamado *stack overflow.*

Notação polonesa (reversa)



Fonte: http://www.quickmeme.com/ e
http://danicollinmotion.com/

PF 6.3

http://www.ime.usp.br/~pf/algoritmos/aulas/pilha.html
http://en.wikipedia.org/wiki/RPN_calculator
http://en.wikipedia.org/wiki/Shunting-yard_algorithm

Notação polonesa

Usualmente os operadores são escritos **entre** os operandos como em

```
(A + B) * D + E / (F + A * D) + C
```

Essa é a chamada **notação infixa**.

Notação polonesa

Usualmente os operadores são escritos **entre** os operandos como em

```
(A + B) * D + E / (F + A * D) + C
```

Essa é a chamada **notação infixa**.

Na **notação polonesa** ou **posfixa**
os operadores são escritos **depois** dos operandos

```
A B + D * E F A D * + / + C +
```

Notação polonesa

Problema: Traduzir para notação posfixa a expressão infixa armazenada em uma cadeia de caracteres `inf`.

Notação polonesa

Problema: Traduzir para notação posfixa a expressão infixa armazenada em uma cadeia de caracteres `inf`. Suponha que na expressão só ocorrem os operadores binários `'+'`, `'-'`, `'*'`, `'/'` além de `'('` e `')'`.

Notação polonesa

Problema: Traduzir para notação posfixa a expressão infixa armazenada em uma cadeia de caracteres `inf`. Suponha que na expressão só ocorrem os operadores binários `'+'`, `'-'`, `'*'`, `'/'` além de `'('` e `')'`.

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))))	ABCDEF+-+-+
A*(B+(C*(D+(E*(F+G)))))	ABCDEFG+*+*+*

Simulação

`inf` = expressão infixa

`s` = pilha

`posf` = expressão posfixa

Simulação

inf = (A*(B*C+D))

inf[0 . . i-1]	s[0 . . t-1]	posf[0 . . j-1]
(	(	
(A	(	A
(A*	(*	A
(A*(	(*(	A
(A*(B	(*(	AB
(A*(B*	(*(*	AB
(A*(B*C	(*(*	ABC
(A*(B*C+	(*(+	ABC*
(A*(B*C+D	(*(+	ABC*D
(A*(B*C+D)	(*	ABC*D+
(A*(B*C+D))		ABC*D+*

Infixa para posfixa

Recebe uma expressão infixa `inf` e
devolve a correspondente expressão posfixa.

```
char *infixaParaPosfixa(char *inf) {

   char *posf; /* expressao polonesa */
   int  n = strlen(inf);
   int  i;  /* percorre infixa */
   int  j;  /* percorre posfixa */
   char *s; /* pilha */
   int  t;  /* topo da pilha */
   char x;  /* item do topo da pilha */

   /*aloca area para expressao polonesa*/
   posf = mallocSafe((n+1)*sizeof(char));
   /* O '+1' eh para o '\0' */
```

cases '(' e ')'

```
s = mallocSafe(n * sizeof(char));
t = 0;

/* examina cada item da infixa */
for (i = j = 0; i < n; i++) {
   switch (inf[i]) {
      case '(':
         s[t++] = inf[i];
         break;
```

cases '(' e ')'

```
s = mallocSafe(n * sizeof(char));
t = 0;

/* examina cada item da infixa */
for (i = j = 0; i < n; i++) {
   switch (inf[i]) {
      case '(':
         s[t++] = inf[i];
         break;

      case ')':
         while ((x = s[--t]) != '(')
            posf[j++] = x;
         break;
```

cases '+', '-', '*', e '/'

```
case '+':
case '-':
   while (t != 0 && (x = s[t-1]) != '(')
      posf[j++] = s[--t];
   s[t++] = inf[i];
   break;
```

cases '+', '-', '*', e '/'

```
case '+':
case '-':
   while (t != 0 && (x = s[t-1]) != '(')
      posf[j++] = s[--t];
   s[t++] = inf[i];
   break;
case '*':
case '/':
   while (t != 0 && (x = s[t-1]) != '('
                  && x != '+' && x != '-')
      posf[j++] = s[--t];
   s[t++] = inf[i];
   break;
```

default e finalizações

```
        default:
            if (inf[i] != ' ')
                posf[j++] = inf[i];
    } /* fim switch */
} /* fim for (i=j=0...)  */
```

default e finalizações

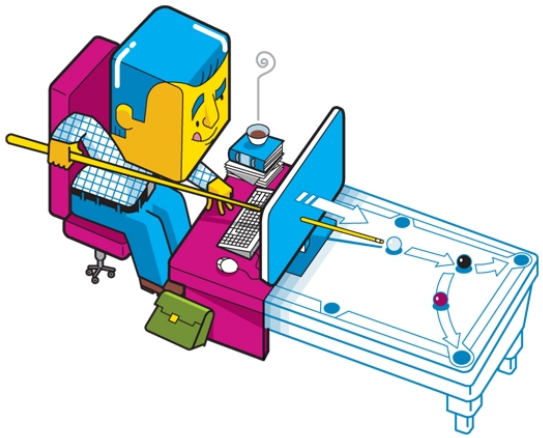
```
        default:
            if (inf[i] != ' ')
                posf[j++] = inf[i];
        } /* fim switch */
    } /* fim for (i=j=0...)  */

    /* desempilha todos os operadores que restaram */
    while (t != 0)
        posf[j++] = s[--t];
    posf[j] = '\0'; /* fim expr polonesa */

    free(s);
    return posf;
} /* fim funcao */
```

Consumo de tempo e espaço

O consumo de tempo da função `infixaParaPosfixa(inf)` é proporcional a `n`, onde `n` é o número de caracteres na string `inf`.

O espaço extra utilizado pela função `infixaParaPosfixa(inf)` é proporcional a `n`, onde `n` é o número de caracteres na string `inf`.

Interfaces



Fonte: http://allfacebook.com/

Before I built a wall I'd ask to know

What I was walling in or walling out,

And to whom I was like to give offence.

Something there is that doesn't love a wall,

That wants it down.

Robert Frost, *Mending Wall*

The Practice of Programming

B.W.Kernigham e R. Pike

S 3.1, 4.2, 4.3, 4.4

Interfaces

Uma **interface** (=*interface*) é uma fronteira entre a implementação de uma biblioteca e o programa que usa a biblioteca.

Um **cliente** (=*client*) é um programa que chama alguma função da biblioteca.

Implementação

```
double sqrt(double x){
  [...]
return raiz;
}
      [...]
```

libm

Interface

```
double sqrt(double);
double sin(double);
double cos(double);
double pow(double,double);
      [...]
```

math.h

Cliente

```
#include <math.h>

      [...]
c = sqrt(a*a+b*b);
      [...]
```

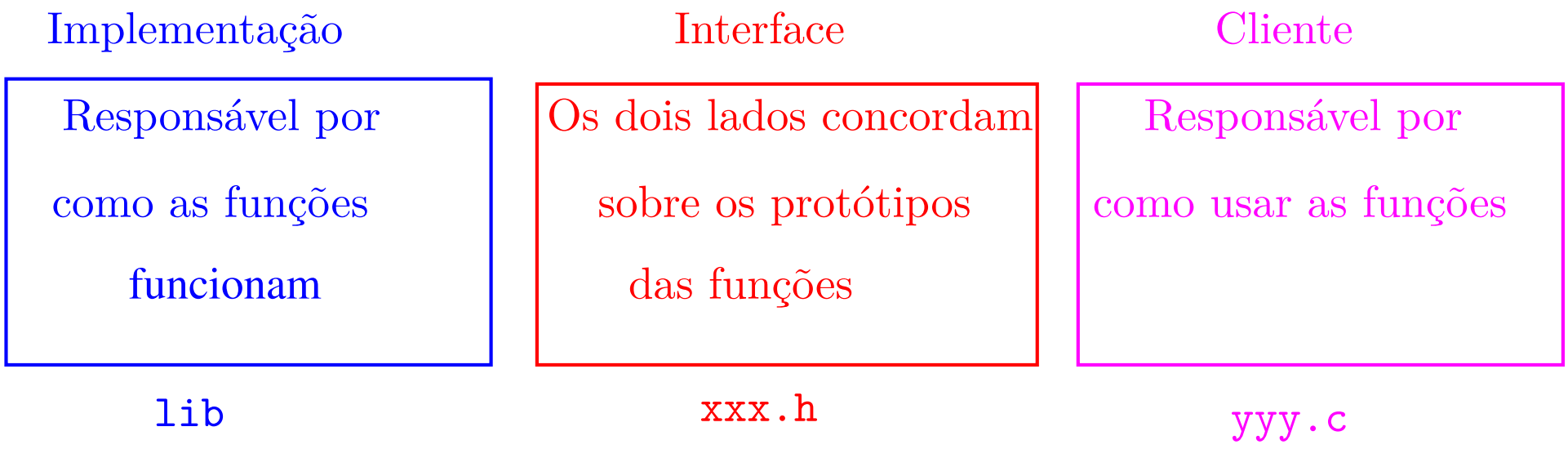
prog.c

Interfaces

Para cada função na biblioteca o cliente precisa saber

- o seu nome, os seus argumentos
  e os tipos desses argumentos;
- o tipo do resultado que é retornado.

Só a quem implementa interessa os detalhes de implementação.



Interfaces

Entre as decisões de projeto estão

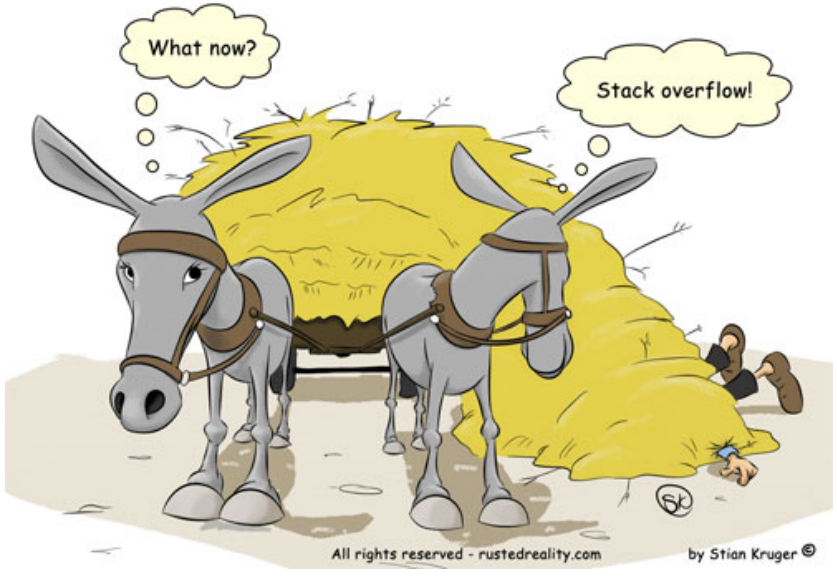
Interface: quais serviços serão oferecidos?
A interface é um "contrato" entre o usuário e o projetista.

Ocultação: qual informação é visível e qual é privada?
Uma interface deve prover acesso aos componente enquanto esconde detalhes de implementação que podem ser alterados sem afetar o usuário.

Recursos: quem é responsável pelo gerenciamento de memória e outros recursos?

Erros: quem detecta e reporta erros e como?

Interfaces para pilhas



Fonte: http://rustedreality.com/stack-overflow/

S 3.1, 4.2, 4.3, 4.4

Interface item.h

```
/* item.h */

#ifndef HEADER_Item

#define HEADER_Item

typedef char Item;

#endif
```

Interface stack.h

```
/*
* stack.h
* INTERFACE: funcoes para manipular uma pilha
*/

#include "item.h"

void stackInit(int);
int stackEmpty();
void stackPush(Item);
Item stackPop();
Item stackTop();
void stackFree();
void stackDump();
```

Infixa para posfixa novamente

Recebe uma expressão infixa `inf` e devolve a correspondente expressão posfixa.

```
char *infixaParaPosfixa(char *inf) {

   char *posf; /* expressao polonesa */
   int  n = strlen(inf);
   int  i;  /* percorre infixa */
   int  j;  /* percorre posfixa */
   char x;  /* item do topo da pilha */

   /*aloca area para expressao polonesa*/
   posf = mallocSafe((n+1)*sizeof(char));
   /* O '+1' eh para o '\0' */
```

cases '(' e ')'

```
stackInit(n) /* inicializa a pilha */
```

cases '(' e ')'

```
stackInit(n) /* inicializa a pilha */

/* examina cada item da infixa */
for (i = j = 0; i < n; i++) {
   switch (inf[i]) {
      case '(':
         stackPush(inf[i]);
         break;

      case ')':
         while((x = stackPop()) != '(')
            posf[j++] = x;
         break;
```

cases '+', '-', '*' e '/'

```
case '+':
case '-':
   while (!stackEmpty()
      && (x = stackTop()) != '(')
      posf[j++] = stackPop();
   stackPush(inf[i]);
   break;
case '*':
case '/':
   while (!stackEmpty()
      && (x = stackTop()) != '('
      && x != '+' && x != '-')
      posf[j++] = stackPop();
   stackPush(inf[i]);
   break;
```

default e finalizações

```
        default:
            if(inf[i] != ' ')
                posf[j++] = inf[i];
        } /* fim switch */
    } /* fim for (i=j=0...)  */

    /* desempilha todos os operandos que restaram */
    while (!stackEmpty())
        posf[j++] = stackPop()
    posf[j] = '\0'; /* fim expr polonesa */
    stackFree();
    return posf;
} /* fim funcao */
```