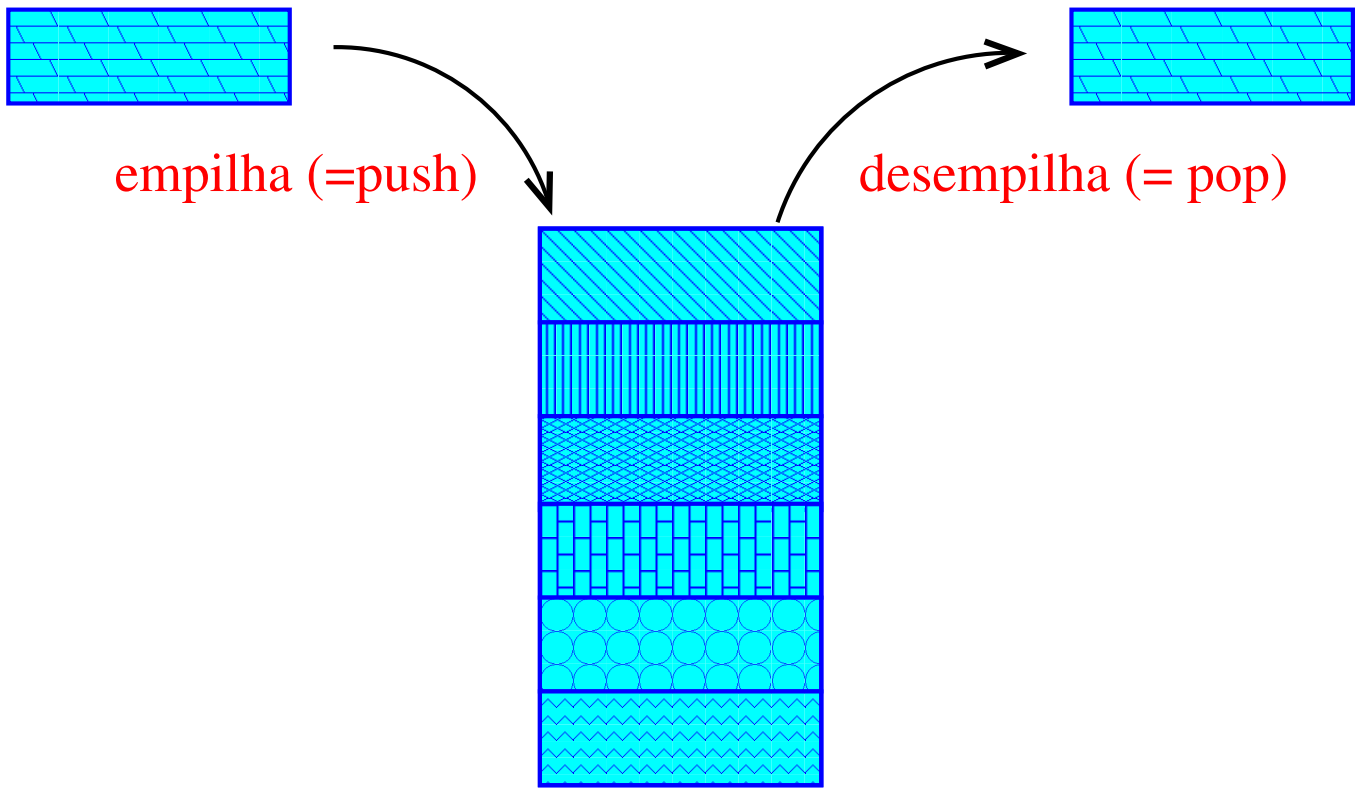
Melhores momentos

AULA 14

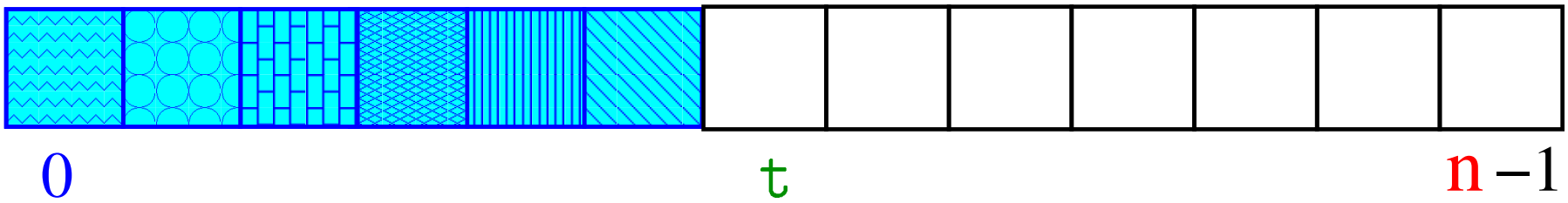
Pilhas

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**.



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`".

AULA 15

Notação polonesa

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**.

Na **notação polonexa** ou **posfixa** os operadores são escritos **depois** do 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`.

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[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 */

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

case '('

```
/* stackInit(n):  inicializa a pilha */
s = (char*) mallocc(n * sizeof(char));
t = 0;
/* examina cada item da infixa */
for (i = j = 0; i < n; i++) {
   switch (inf[i]) {
        char x; /* item do topo da pilha */
        case '(':
            /* stackPush(infixa[i]) */
            s[t++] = inf[i];
            break;
```

case ')'

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

case '+', case '-'

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

case '*', case '/'

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

default

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

Finalizações

```
    /* desempilha todos os operandos que
        restaram */
    /* !stackEmpty() */
    while (t != 0)
        posf[j++] = s[--t]; /* stackPop() */
    posf[j] = '\0'; /* fim expr polonesa */
    /* stackFree() */
    free(s);
    return posf;
} /* fim funcao */
```

Interfaces

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.

The Practice of Programming

B.W.Kernigham e R. Pike

S 3.1

Interfaces

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

Um **cliente** (=*interface*) 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);
      [...]
```

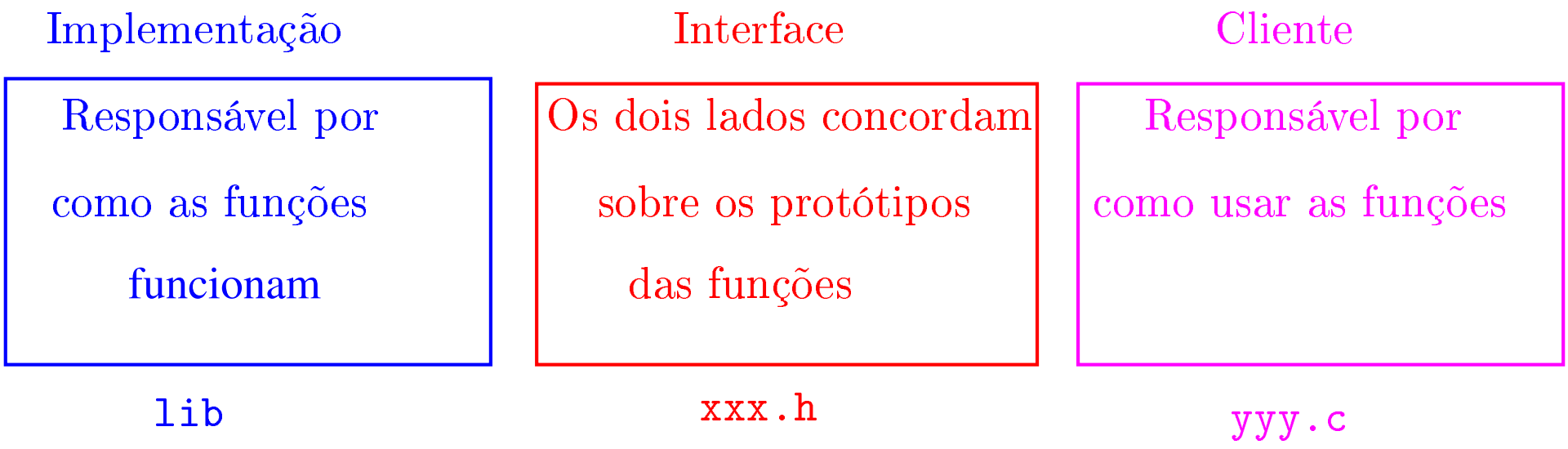
prog.c

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?

Interface item.h

```
/* Item.h */
typedef char Item;
```

Interface stack.h

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

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

case '('

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

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

case ')'

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

case '+', case '-'

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

case '*', case '/'

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

default

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

Finalizações

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

Implementação stack.c

```
#include <stdlib.h>
#include <stdio.h>
#include "item.h"
/*
 * PILHA: implementacao em vetor
 */
static char *s; /* pilha */
static int t;
/* t eh o indice do topo da pilha, s[t]
 * eh a 1a.  posicao vaga da pilha
 */
```

Implementação stack.c

```
void
stackInit(int n)
{
   s = (Item*) malloc(n*sizeof(Item));
   t = 0;
}

int
stackEmpty()
{
   return t == 0;
}
```

Implementação stack.c

```
void
stackPush(Item item)
{
   s[t++] = item;
}

Item
stackPop()
{
   return s[--t];
}
```

Implementação stack.c

```
Item
stackTop()
{
   return s[t-1];
}

void
stackFree()
{
   free(s);
}
```

Implementação stack.c

```
void
stackDump()
{
   int k;

   fprintf(stdout,"pilha :  ");
   if (t == 0) fprintf(stdout, "vazia.");
   for (k = 0; k < t; k++)
       fprintf(stdout, "%c ", s[k]);
   fprintf(stdout,"\n");
}
```

Compilação

cria o obj `stack.o`

```
> gcc -Wall -O2 -ansi -pedantic -Wno-unused-result -c stack.c
```

cria o obj `polonesa.o`

```
> gcc -Wall -O2 -ansi -pedantic -Wno-unused-result
    -c polonesa.c
```

cria o executável `polonesa`

```
> gcc stack.o polonesa.o -o polonesa
```

Makefile

Hmmm. Ler o tópico `Makefile` no fórum.

```
polonesa: polonesa.o stack.o
        gcc polonesa.o stack.o -o polonesa

polonesa.o: polonesa.c
        gcc -Wall -O2 -ansi -pedantic \
        -Wno-unused-result -c polonesa.c

stack.o: stack.c item.h
        gcc -Wall -O2 -ansi -pedantic \
        -Wno-unused-result -c stack.c
```