

AULA 13

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, 4.2, 4.3, 4.4

Interfaces

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

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.

Implementação

Responsável por
como as funções
funcionam

lib

Interface

Os dois lados concordam
sobre os protótipos
das funções

xxx.h

Cliente

Responsável por
como usar as funções

yyy.c

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

S 3.1, 4.2, 4.3, 4.4

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 infixada `inf` e devolve a correspondente expressão `posfixa`.

```
char *infixaParaPosfixa(char *inf) {  
    char *posf; /* expressao polonesa */  
    int n = strlen(inf);  
    int i; /* percorre infixada */  
    int j; /* percorre posfixa */  
  
    /*aloca area para expressao polonesa*/  
    posf = mallocSafe((n+1)*sizeof(char));  
    /* 0 '+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.

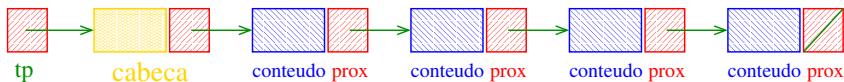
```
polonesa: polonesa.o stack.o
    gcc polonesa.o stack.o -o polonesa
```

```
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
```

Pilha implementada em uma lista encadeada

A pilha será armazenada em uma **lista encadeada** com **cabeça**.



O ponteiro **tp** aponta para a **cabeça** da lista.

tp->**prox**->**conteudo** é o elemento do **topo** da pilha.

A pilha está **vazia** se "**tp**->**prox** == **NULL**".

A pilha está **cheia** se ... acabou a memória disponível.

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();
```


Implementação stack.c

```
#include <stdlib.h>
#include <stdio.h>
#include "item.h"
/*
 * PILHA: implementacao em lista encadeada
 */
typedef struct stackNode* Link;
struct stackNode{
    Item conteudo;
    Link prox;
};
static Link tp;
```

Implementação stack.c

```
void
stackInit(int n)
{
    tp = mallocSafe(sizeof *tp);
    tp->prox = NULL;
}
```

```
int
stackEmpty()
{
    return tp->prox == NULL;
}
```

Implementação stack.c

```
void  
stackPush(Item item)  
{  
    Link nova = mallocSafe(sizeof *nova);  
  
    nova->conteudo = item;  
    nova->prox = tp->prox;  
    tp->prox = nova;  
}
```

Implementação stack.c

Item

stackPop()

{

Link p = tp->prox;

Item conteudo = p->conteudo;

tp->prox = p->prox;

free(p);

return conteudo;

}

Implementação stack.c

Item

```
stackTop()
```

```
{
```

```
    return tp->prox->conteudo;
```

```
}
```

Implementação stack.c

```
void
stackFree()
{
    while (tp != NULL)
    {
        Link p = tp;
        tp = p->prox;
        free(p);
    }
}
```

Implementação stack.c

```
void
stackDump() {
    int p = tp->prox;

    fprintf(stdout, "pilha : ");
    if (p==NULL) fprintf(stdout, "vazia.");
    while (p != NULL) {
        fprintf(stdout, "%c ", p->conteudo);
        p = p->prox;
    }
    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 -o polonesa stack.o polonesa.o
```


Makefile

Hmmm. Ler o tópico **Makefile** no fórum.

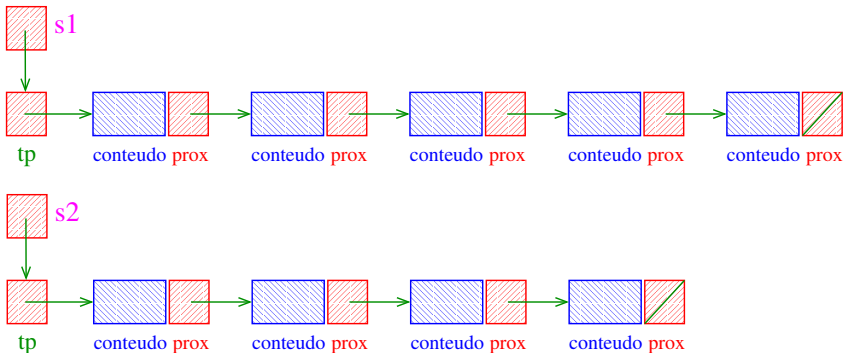
```
polonesa: polonesa.o stack.o
    gcc polonesa.o stack.o -o polonesa
```

```
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
```

PilhaS implementadaS em listaS encadeadaS

As pilhas serão armazenada em **listaS encadeadaS** sem **cabeça**.



PilhaS implementadaS em listaS encadeadaS

Para **cada pilha** há um ponteiro **tp** para a lista .

tp->conteudo é o elemento do **topo** da pilha .

Uma pilha **s** está **vazia** se “**s->tp == NULL**” .

Uma pilha está **cheia** se ... acabou a memória disponível.

Interface stack.h

```
/*  
 * stack.h  
 * INTERFACE: funcoes para manipular uma  
 * pilha  
 */  
typedef struct stack *Stack;  
Stack stackInit(int);  
int stackEmpty(Stack);  
void stackPush(Stack, Item);  
Item stackPop(Stack);  
Item stackTop(Stack);  
void stackFree(Stack);  
void stackDump(Stack);
```

Infixa para posfixa novamente...

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

```
char *infixaParaPosfixa(char *inf) {
    char *posf; /* expressao polonesa */
    int n = strlen(inf);
    int i; /* percorre infixada */
    int j; /* percorre posfixa */
    Stack s; /* PILHA */

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

case '('

```
s = 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(s, inf[i]);  
            break;
```

case ')')

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

case '+', case '-'

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


case '*', case '/'

```
case '*':  
case '/':  
    while (!stackEmpty()  
           && (x = stackTop(s)) != '('  
           && x != '+' && x != '-')  
        posf[j++] = stackPop(s);  
    stackPush(s, 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(s))
    posf[j++] = stackPop(s)
posf[j] = '\0'; /* fim expr polonesa */
stackFree(s);
return posf;
} /* fim funcao */
```

Implementação stack.c

```
#include "item.h"
/* PILHA: implementacao em lista encadeada
 */
typedef struct stackNode* Link;
struct stackNode{
    Item conteudo;
    Link prox;
};
struct stack {
    Link tp;
};
typedef struct stack *Stack;
```

Implementação stack.c

Stack

```
stackInit(int n)
{
    Stack s = mallocSafe(sizeof *s);

    s->tp = NULL;
    return s;
}
```

Implementação stack.c

```
int  
stackEmpty(Stack s)  
{  
    return s->tp == NULL;  
}
```

Implementação stack.c

```
void  
stackPush(Stack s, Item item)  
{  
    Link nova = mallocSafe(sizeof *nova);  
  
    nova->conteudo = item;  
    nova->prox = s->tp;  
    s->tp = nova;  
}
```

Implementação stack.c

Item

```
stackPop(Stack s)
```

```
{
```

```
    Link p = s->tp;
```

```
    Item conteudo = p->conteudo;
```

```
    s->tp = p->prox;
```

```
    free(p);
```

```
    return conteudo;
```

```
}
```


Implementação stack.c

Item

```
stackTop(Stack s)
```

```
{
```

```
    return s->tp->conteudo;
```

```
}
```

Implementação stack.c

```
void
stackFree(Stack s)
{
    while (s->tp != NULL)
    {
        Link p = s->tp;
        s->tp = p->prox;
        free(p);
    }
    free(s);
}
```

Implementação stack.c

```
void
stackDump() {
    int p = s->tp;

    fprintf(stdout, "pilha : ");
    if (p==NULL) fprintf(stdout, "vazia.");
    while (p != NULL) {
        fprintf(stdout, "%c ", p->conteudo);
        p = p->prox;
    }
    fprintf(stdout, "\n");
}
```