

## AED1 - Aula 15

### Bibliotecas e interfaces, pilhas, leiaute da memória

Uma biblioteca é uma coleção de funções

- que podem ser utilizadas por programas.

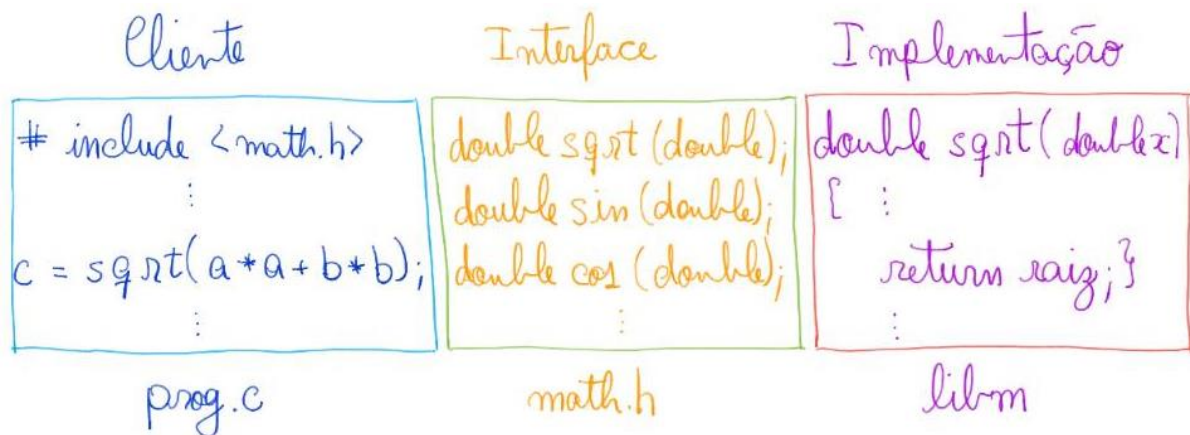
Um cliente é um programa que utiliza alguma função de uma biblioteca.

Uma interface é a fronteira entre

- a implementação de uma biblioteca
  - e os clientes que a utilizam.

Para cada função na biblioteca,

- o cliente precisa conhecer sua assinatura, i.e.,
  - o nome da função,
  - seus argumentos (e tipos),
  - e o tipo do resultado devolvido.
- Já os detalhes de implementação
  - não são relevantes para o cliente.



Algumas decisões de projeto envolvendo bibliotecas.

- Interface
  - Quais as funções oferecidas?
- Ocultação
  - Quais informações são públicas e quais são privadas?
- Recursos
  - Quem é responsável por gerenciar memória e outros recursos?
- Erros
  - Quem é responsável por detectar e reportar erros?

Vamos ver como implementar nossa própria biblioteca para pilha.

- Segue o código da interface pilha.h:

```
typedef struct pilha Pilha;
```

```

Pilha *criaPilha();
void empilha(Pilha *s, char x);
char desempilha(Pilha *s);
char consultaTopo(Pilha *s);
int pilhaVazia(Pilha *s);
void imprimePilha(Pilha *s);
int tamPilha(Pilha *s);
Pilha *liberaPilha(Pilha *s);

```

Para usar essa biblioteca,

- é necessário incluir uma chamada para ela no início do seu programa.

```
#include "pilha.h"
```

Vamos usar essa biblioteca para

- relembrar algumas operações básicas com pilhas.

```

int main(int argc, char *argv[])
{
    char *inf, *posf;
    Pilha *s;
    char x;

    if (argc != 2)
    {
        printf("Numero incorreto de parametros! Ex.: .\\usaPilha
\\(A+B*(C-D)+E)\\");
        return 0;
    }
    inf = argv[1];

    /* inicializa a pilha */
    s = criaPilha();

    /* empilha abc */
    empilha(s, 'a');
    empilha(s, 'b');
    empilha(s, 'c');

```

```

    /* imprime pilha */
    imprimePilha(s);

    /* desempilha e armazena em x */
    x = desempilha(s);
    printf("%c\n", x);

    /* consulta topo da pilha */
    printf("%c\n", consultaTopo(s));

    /* imprime pilha */
    imprimePilha(s);

    /* tamanho da pilha */
    printf("%d\n", tamPilha(s));

    /* libera a pilha */
    s = liberaPilha(s);

    printf("Infixa = %s\n", inf);
    posf = infix2posfix(inf);
    printf("Posfixa = %s\n", posf);

    return 0;
}

```

- além de resolver/revisitar
  - o problema da conversão de notação infixa para pósfixa.

```

// Esta função recebe uma expressão infixa inf
// e devolve a correspondente expressão posfixa.
char *infix2posfix(char *inf)
{
    int n = strlen(inf);
    char *posf; // expressão pósfixa
    posf = malloc((n + 1) * sizeof(char));
    int i;      // percorre infixa
    int j;      // percorre posfixa
    Pilha *s;   // pilha

```

```

// inicializa a pilha
s = criaPilha();

for (i = j = 0; inf[i] != '\0'; i++)
{
    switch (inf[i])
    {
        char x; // auxiliar para item do topo da pilha
        case '(':
            empilha(s, inf[i]); // empilha
            break;
        case ')':
            // move da pilha pra pósfixa até
encontrar '('
            x = desempilha(s); // desempilha
            while (x != '(')
            {
                posf[j++] = x;
                x = desempilha(s); // desempilha
            }
            break;
        case '+':
        case '-':
            // joga na pósfixa conteúdo da pilha até esta ficar
            // vazia ou encontrar o início do bloco '('
            while (!pilhaVazia(s) && consultaTopo(s) != '(')
            {
                posf[j++] = desempilha(s); // desempilha
            }
            empilha(s, inf[i]); // empilha
            break;
        case '*':
        case '/':
            // joga na pósfixa conteúdo da pilha até esta ficar
            // vazia, encontrar o início do bloco '(', ou
            // encontrar operador de menor precedência '+' ou '-'

```

```

        while (!pilhaVazia(s) && ((x = consultaTopo(s)) != '('
&& x != '+' && x != '-'))
        {
            posf[j++] = desempilha(s); // desempilha
        }
        empilha(s, inf[i]);
        break;
    default:
        if (inf[i] != ' ')
            posf[j++] = inf[i];
    }
}
// desempilha o que sobrou na pilha
while (!pilhaVazia(s))
    posf[j++] = desempilha(s);
posf[j] = '\0';
s = liberaPilha(s);
return posf;
}

```

A seguir temos a implementação da biblioteca usando vetor.

```

#include <stdio.h>
#include <stdlib.h>

#include "pilha.h"

#define TAM_MAX 100

struct pilha
{
    char *vetor;
    int topo;
};

Pilha *criaPilha()
{
    Pilha *s;

```

```

    s = (Pilha *)malloc(sizeof(Pilha));
    s->vetor = (char *)malloc(TAM_MAX * sizeof(char));
    s->topo = 0;
    return s;
}

void empilha(Pilha *s, char x)
{
    s->vetor[s->topo] = x;
    (s->topo)++;
}

char desempilha(Pilha *s)
{
    (s->topo)--;
    return s->vetor[s->topo];
}

char consultaTopo(Pilha *s)
{
    return s->vetor[(s->topo) - 1];
}

int pilhaVazia(Pilha *s)
{
    return s->topo <= 0;
}

int pilhaCheia(Pilha *s)
{
    return s->topo == TAM_MAX;
}

void imprimePilha(Pilha *s)
{
    for (int i = (s->topo) - 1; i >= 0; i--)
        printf("%c ", s->vetor[i]);
}

```

```

    printf("\n");
}

int tamPilha(Pilha *s)
{
    return s->topo;
}

Pilha *liberaPilha(Pilha *s)
{
    free(s->vetor);
    free(s);
    return NULL;
}

```

A seguir temos a implementação da biblioteca usando lista encadeada.

```

#include <stdio.h>
#include <stdlib.h>

#include "pilha.h"

typedef struct celula
{
    char conteudo;
    struct celula *prox;
} Celula;

struct pilha
{
    Celula *lst;
    int tam;
};

Pilha *criaPilha()
{
    Pilha *s;
    s = (Pilha *)malloc(sizeof(Pilha));
}

```

```

    s->lst = NULL;
    s->tam = 0;
    return s;
}

void empilha(Pilha *s, char x)
{
    Celula *nova;
    nova = malloc(sizeof(Celula));
    nova->conteudo = x;
    nova->prox = s->lst;
    s->lst = nova;
    s->tam++;
}

char desempilha(Pilha *s)
{
    char x;
    Celula *morta;
    morta = s->lst;
    x = morta->conteudo;
    s->lst = morta->prox;
    free(morta);
    morta = NULL;
    (s->tam)--;
    return x;
}

char consultaTopo(Pilha *s)
{
    return s->lst->conteudo;
}

int pilhaVazia(Pilha *s)
{
    return s->lst == NULL;
}

```



```
int pilhaCheia(Pilha *s)
{
    return 0;
}

void imprimePilha(Pilha *s)
{
    Celula *p;
    p = s->lst;
    while (p != NULL)
    {
        printf("%c ", p->conteudo);
        p = p->prox;
    }
    printf("\n");
}

int tamPilha(Pilha *s)
{
    return s->tam;
}

Pilha *liberaPilha(Pilha *s)
{
    Celula *p, *morta;
    p = s->lst;
    while (p != NULL)
    {
        morta = p;
        p = p->prox;
        free(morta);
    }
    free(s);
    return NULL;
}
```

## Compilando biblioteca

Para implementar e compilar um programa que usa nossa biblioteca,

- primeiro incluímos uma chamada para ela no início do programa,

```
#include "pilha.h"
```

- então compilamos a biblioteca em um programa objeto  
"gcc -c pilha.c" ou  
"gcc -Wall -O2 -pedantic -Wno-unused-result -c pilha.c"
- e, finalmente, compilamos o programa principal usando esse programa objeto  
"gcc pilha.o usaPilha.c -o usaPilha" ou  
"gcc -Wall -O2 -pedantic -Wno-unused-result pilha.o usaPilha.c -o usaPilha"

Também podemos compilar o programa principal em um programa objeto

```
"gcc -c usaPilha.c" ou
```

```
"gcc -Wall -O2 -pedantic -Wno-unused-result -c usaPilha.c"
```

- e então compilar os dois programas objetos no executável  
"gcc pilha.o usaPilha.o -o usaPilha"

Ou, no extremo oposto, compilar tudo diretamente, sem usar programas objeto

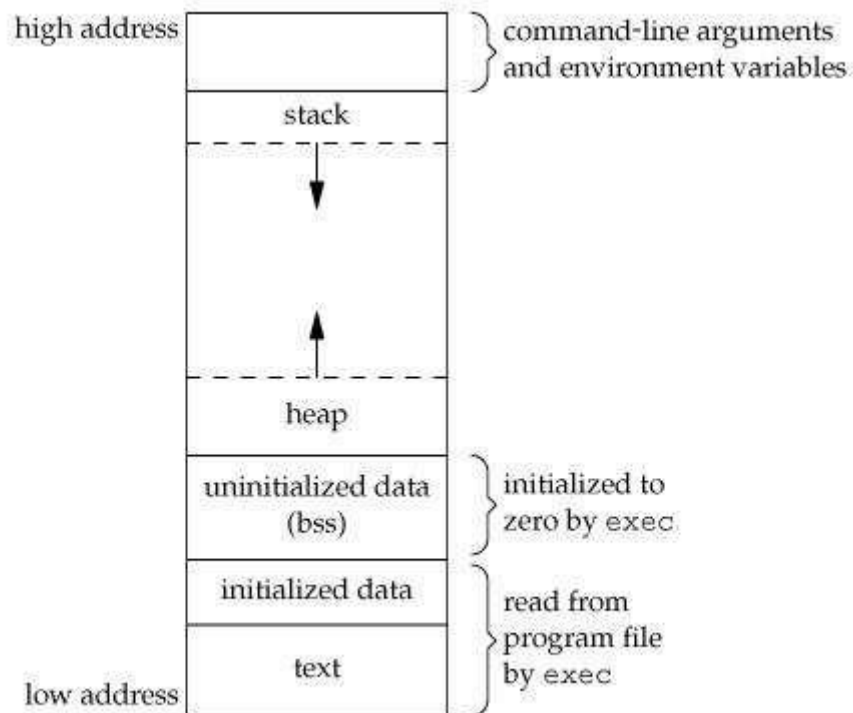
```
"gcc pilha.c usaPilha.c -o usaPilha" ou
```

```
"gcc -Wall -O2 -pedantic -Wno-unused-result pilha.c usaPilha.c -o usaPilha"
```

Observe que, como nossas duas implementações de pilha usam a mesma interface,

- podemos trocar a biblioteca utilizada
  - sem afetar o funcionamento do nosso programa de teste.

## Leitura da memória de um programa em C



Fonte: <http://cs-fundamentals.com/c-programming/memory-layout-of-c-program-code-data-segments.php/>

Breve descrição de cada segmento de memória:

- **stack (pilha)**: variáveis locais, parâmetros de funções, endereços de retorno.
- **heap (não a estrutura de dados)**: memória alocada dinamicamente, administrada por `malloc()` e `free()`.
- **uninitialized data (bss)**: variáveis estáticas (`static`) e globais não inicializadas (`static int i;`).
- **initialized data**: variáveis estáticas (`static`) inicializadas (`static int i = 0;`).
- **text**: código do programa (para onde apontam os endereços de retorno).

Exemplo comando “size”:

“size usaPilha.exe”

text	data	bss	dec	hex	filename
16140	1572	112	17824	45a0	usaPilha.exe