

Estrutura de Dados

Parte III – Pilhas: implementação
com ponteiros

Pilhas com Alocação Dinâmica de Memória

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

//Definição da Estrutura .....

typedef struct TipoElemento {
    int valor;
    TipoElemento *acima, *abaixo;
}TElemento;

typedef struct TipoPilha {
    TElemento *topo, *base;
}TPilha;

//Prototipos das funções .....

void inicPilha(TPilha *p);
int pilhaVazia(TPilha *p);
void empilha(TPilha *p, int numero);
TElemento *desempilha(TPilha *p);
int elemTopo(TPilha *p);
int menu();
```

Pilhas com Alocação Dinâmica de Memória

```
//Corpo PRINCIPAL (main) do Programa .....

int main() {
    int op;
    TPilha pilha;
    inicPilha(&pilha);

    do {
        op = menu();

        switch(op) {
            case 1: {
                int numero;
                printf("\n\n\tEMPILHAR o valor: ");
                scanf("%d", &numero);
                empilha(&pilha, numero);
                break;
            }
            case 2: {
                TElemento *descartado = desempilha(&pilha);
                if (descartado != NULL) {
                    printf("\n\n\tELEMENTO desempilhado: %d.\n\n", descartado->valor);
                    free(descartado);
                } else {
                    printf("\n\n\tNENHUM ELEMENTO foi desempilhado, pois a PILHA");
                    printf(" encontra-se VAZIA\n\n");
                }
                //if
                system("PAUSE");
                break;
            }
            case 3: {
                printf("\n\n\tO ELEMENTO atualmente no TOPO da PILHA:");
                printf(" %d.\n\n", elemTopo(&pilha));
                system("PAUSE");
                break;
            }
        } //switch

    } while (op != 0);

    return (0);
} //main()
```

Pilhas com Alocação Dinâmica de Memória

```
//Funções .....  
  
void inicPilha(TPilha *p){  
    p->topo = NULL;  
    p->base = NULL;  
} //inicPilha()  
//.....  
int pilhaVazia(TPilha *p){  
    if (p->topo == NULL)  
        return 1;  
    else  
        return 0;  
} //pilhaVazia()  
//.....  
void empilha(TPilha *p, int numero){  
    TElemento *novo = (TElemento *)malloc(sizeof(TElemento));  
    novo->valor = numero;  
  
    if (pilhaVazia(p)){  
        p->topo = novo;  
        p->base = novo;  
  
        novo->abaixo = NULL;  
    } else {  
        novo->valor = numero;  
  
        p->topo->acima = novo;  
        novo->abaixo = p->topo;  
        p->topo = novo;  
  
    } //if...else  
    novo->acima = NULL;  
} //empilha()  
//.....
```

Pilhas com Alocação Dinâmica de Memória

```
TElemento *desempilha(TPilha *p){
    TElemento *desempilhado = NULL;

    if (!pilhaVazia(p)){

        desempilhado = p->topo;

        p->topo = p->topo->abaixo;

        if (p->topo == NULL){
            p->base = NULL;
        } else {
            p->topo->acima = NULL;
        } //if
    } //if
    return desempilhado;
} //desempilha()
//.....
int elemTopo(TPilha *p){
    int valor = -9999;
    if (pilhaVazia(p)){
        printf("\n\nOPERACAO INVALIDA !!!\nPilha encontra-se VAZIA\n");
        printf("Impossivel exibir valor do TOPO da Pilha.\n\n");
        system("PAUSE");
        exit(1);
    } else{
        valor = p->topo->valor;
    } //if
    return valor;
} //elemTopo()
//.....
```

Pilhas com Alocação Dinâmica de Memória

```
// .....  
int menu(){  
    int opcao;  
    TPilha pilha;  
    system("CLS");  
    printf("\n\n\n\t\t\t====| MENU |====\n\n");  
    printf("\t\tOpcoes de selecao:\n");  
    printf("\t0 - Sair (Encerrar Aplicativo).\n\n");  
    printf("\t1 - Empilhar (PUSH).\n");  
    printf("\t2 - Desempilhar (POP).\n");  
    printf("\t3 - Consultar ELEMENTO existente no TOPO da PILHA.\n\n");  
    printf("\t\tOpcao desejada: ");  
    scanf("%d", &opcao);  
  
    if ((opcao < 0) || (opcao > 3)){  
        printf("ERRO:   OPCAO selecionada eh INVALIDA.\n");  
        printf("Suas opcoes limitam-se ao intervalo de 0 (ZERO) a 3 (TRES) -");  
        printf(" INCLUSIVE os valores de ambas extremidades.\n\n");  
        system("PAUSE");  
    }  
    //if  
  
    return opcao;  
}  
//menu()  
// .....
```

Entendendo o Código

```
typedef struct TipoElemento {  
    int valor;  
    TipoElemento *acima, *abaixo;  
}TElemento;
```

```
typedef struct TipoPilha {  
    TElemento *topo, *base;  
}TPilha;
```

TPilha pilha

topo
base

TElemento

valor	acima	abaixo
-------	-------	--------

Entendendo o Código

- Os campos topo e base da estrutura **TPilha** são ponteiros para **TElemento**.
- Ambos **apontam** para uma área de memória que armazene uma estrutura de **TElemento**.
- TElemento apresenta três campos:
 - valor (do tipo inteiro);
 - acima (ponteiro para TElemento);
 - abaixo (ponteiro para TElemento).

Entendendo o Código

- Existe uma variável global de nome **pilha** que é do tipo **TPilha**.
- Por enquanto não há uma variável de tipo **TElemento**.
- A função **inicPilha()** é responsável por deixar a **pilha vazia** – pronta para que comecemos a empilhar **variáveis anônimas** do tipo **TElemento**.

Entendendo inicPilha()

```
//Funções .....
```

```
void inicPilha(TPilha *p){  
    p->topo = NULL;  
    p->base = NULL;  
} //inicPilha()  
//.....
```

TPilha pilha

topo NULL
base NULL

A passagem de parâmetros é por referência: p é um ponteiro para TPilha.

No programa principal (**main()**) a chamada a essa função é **inicPilha(&pilha)**.

Note que **pilha** é a variável global declarada anteriormente (do tipo **TPilha**).

O **&** à esquerda do nome da variável **pilha** significa que o endereço da variável global está sendo repassado para a função. O parâmetro **p** recebe esse endereço na função.

Entendendo inicPilha()

- O comando “**p->topo = NULL;**” deve ser lido da seguinte maneira: “**o campo topo, apontado por p, recebe NULO**”.
- O valor “**NULO**” significa que o **ponteiro topo** não aponta para nenhum endereço de memória.
- O mesmo ocorre com base, que não aponta para nenhum endereço de memória.

Entendendo pilhaVazia()

```
//.....  
int pilhaVazia(TPilha *p){  
    if (p->topo == NULL)  
        return 1;  
    else  
        return 0;  
} //pilhaVazia()  
//.....
```

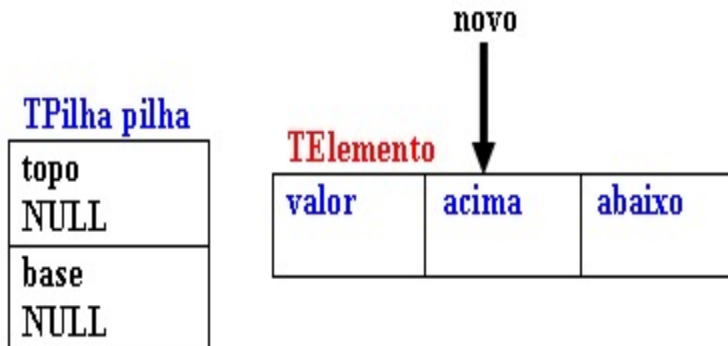
TPilha pilha	
topo	NULL
base	NULL

Se o campo **topo** de **pilha** estiver **NULO** (= = NULL) a **pilha** encontra-se **vazia** e a função **pilhaVazia()** **retorna** o valor **1** (qualquer valor diferente de zero é verdadeiro).

Caso contrário – se **topo** não for **NULO**, mas apontar para algum endereço – a **função retorna 0** (= = FALSE).

Entendendo empilha()

```
void empilha(TPilha *p, int numero){  
    TElemento *novo = (TElemento *)malloc(sizeof(TElemento));
```



A função **empilha()** não retorna qualquer valor (**void**), mas recebe **dois parâmetros de entrada**:

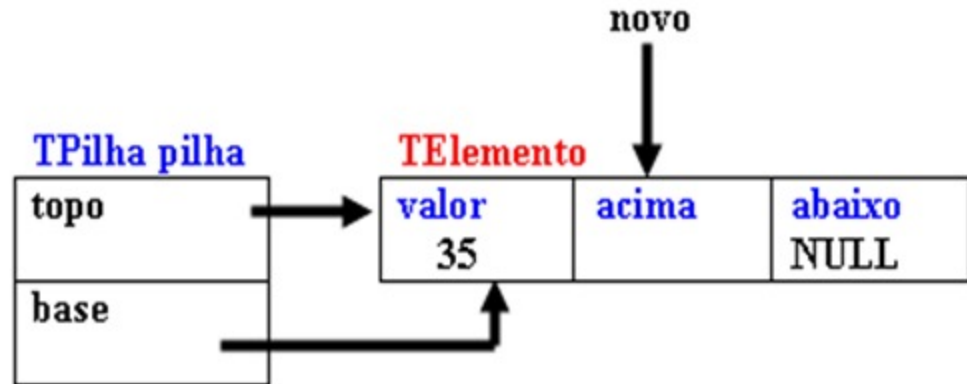
- (1º) p que é um ponteiro para TPilha, e
- (2º) numero que é variável do tipo inteiro.

Uma possível **chamada** dessa **função** a partir do **main()** poderia ser **empilha(&pilha, 35)**.

O comando **malloc** aloca dinamicamente um **segmento de memória** e armazena seu **endereço** no **ponteiro novo** (que aponta para **TElemento**).

Entendendo empilha()

```
    novo->valor = numero;  
  
    if (pilhaVazia(p)) {  
        p->topo = novo;  
        p->base = novo;  
  
        novo->abaixo = NULL;  
    } else {
```



O campo **valor** – apontado por **novo** – recebe **35**.

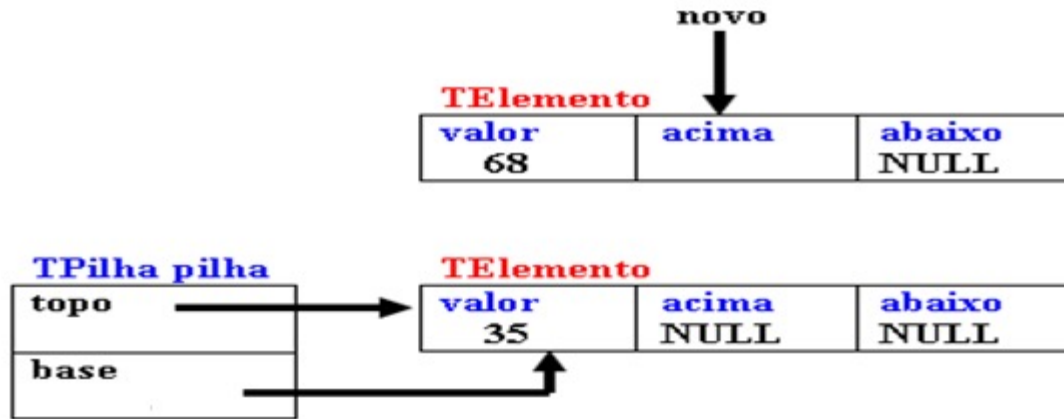
Se a **pilha** apontada por **p** estiver **vazia**:

O campo **topo** apontado por **p** recebe **novo** (aponta para **novo**).

O campo **base** apontado por **p** recebe **novo** (aponta para **novo**).

O campo **abaixo** apontado por **novo** recebe **NULL**.

Entendo empilha()



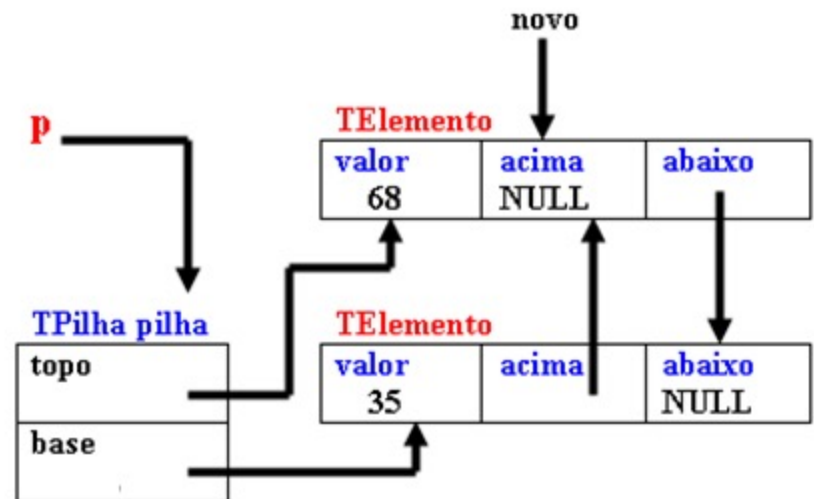
Agora suponha que a **função** foi concluída (o campo **acima** anteriormente apontado por **novo** recebe **NULL**) e **novo** deixa de apontar para o **elemento** de valor **35**.

Então ocorre uma outra **chamada** à **função**: “**empilha(&pilha, 68);**”.

Um **novo segmento de memória** é **alocado** junto ao sistema operacional e seu **endereço inicial** é armazenado no **ponteiro novo**.

Entendendo empilha()

```
    } else {  
  
        p->topo->acima = novo;  
        novo->abaixo = p->topo;  
        p->topo = novo;  
  
    } //if...else  
    novo->acima = NULL;  
  
} //empilha()  
//
```



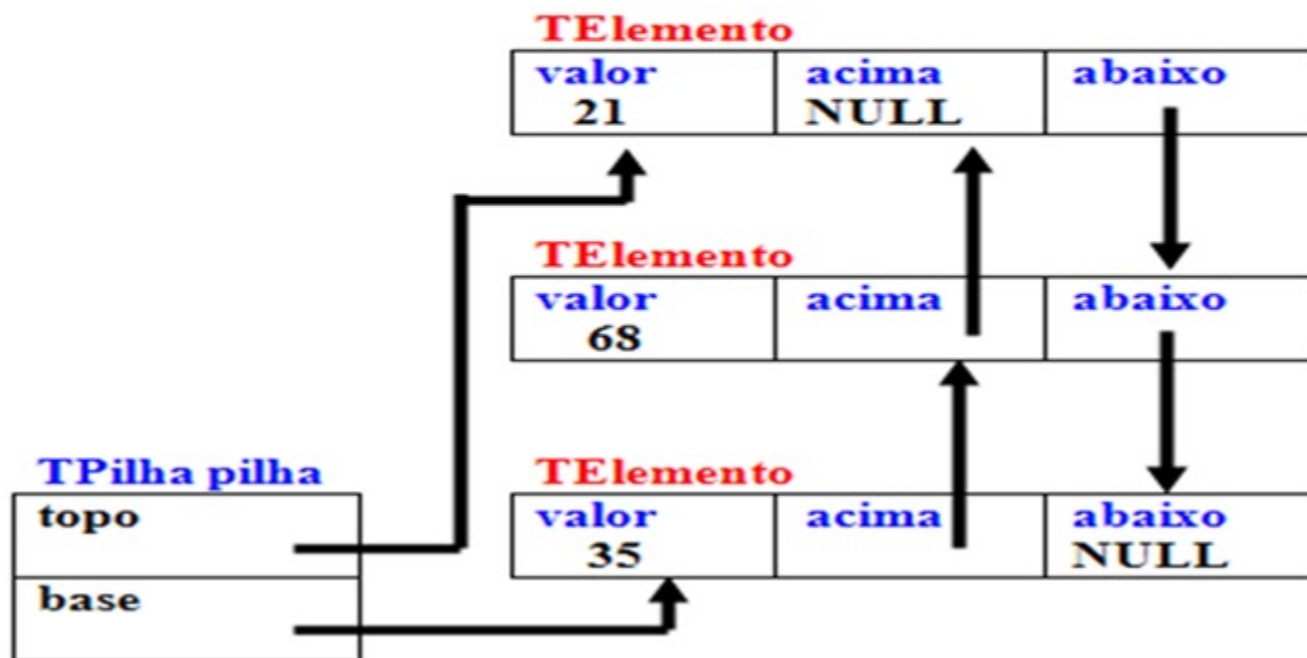
O campo **acima** apontado por **topo** (por sua vez apontado por **p**) – o elemento de valor **35** – aponta para **novo**.

O campo **abaixo** apontado por **novo** (elemento de valor **68**) aponta para o mesmo endereço que **topo** apontado por **p** (elemento de valor **35**).

O campo **topo** apontado por **p** deixa de apontar para elemento de valor **35** e passa a apontar para o elemento apontado por **novo** (de valor **68**).

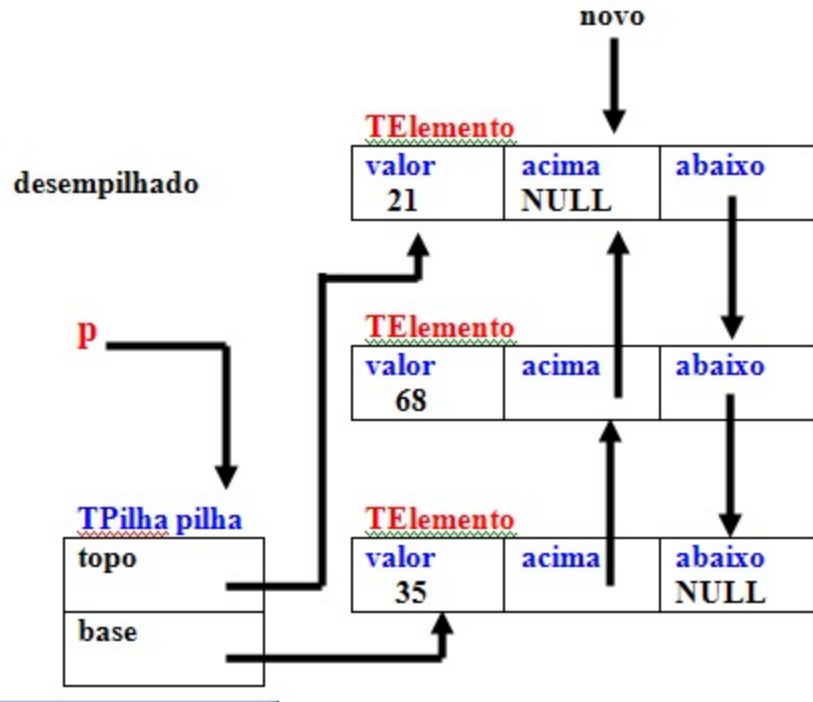
O campo **acima** apontado por **novo** (elemento de valor **68**) recebe **NULL** (não aponta para ninguém).

Situação Hipotética para a Pilha



Entendendo desempilha()

```
// .....  
TElemento *desempilha(TPilha *p){  
    TElemento *desempilhado = NULL;
```

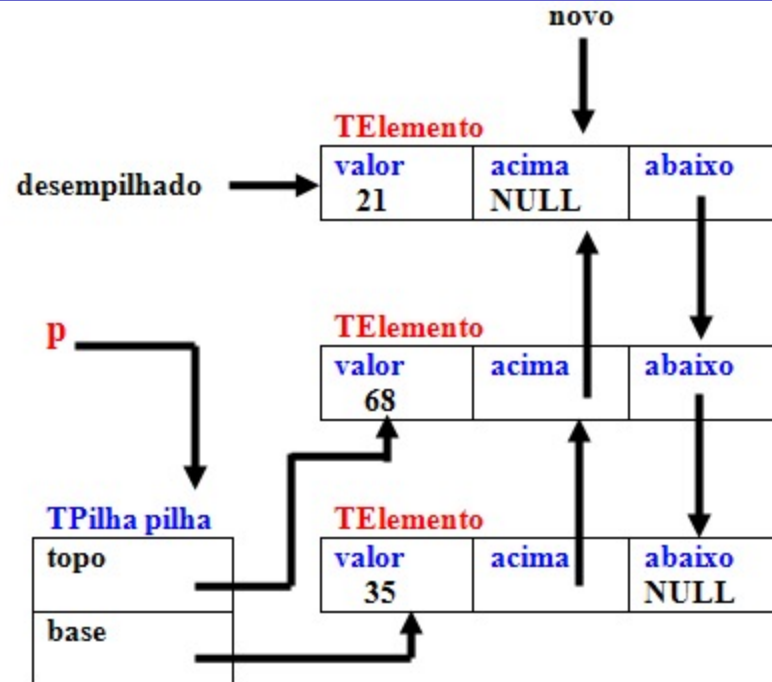


A função **desempilha()** retorna um **ponteiro** para **TElemento** e recebe como **parâmetro de entrada** o **ponteiro p** (que aponta para **TPilha**).

Um **ponteiro** para **TElemento** é declarado com o nome **desempilhado** e inicializado com o valor **NULL** (não aponta para ninguém).

Entendendo desempilha()

```
if (!pilhaVazia(p)){  
  
    desempilhado = p->topo;  
  
    p->topo = p->topo->abaixo;  
}
```



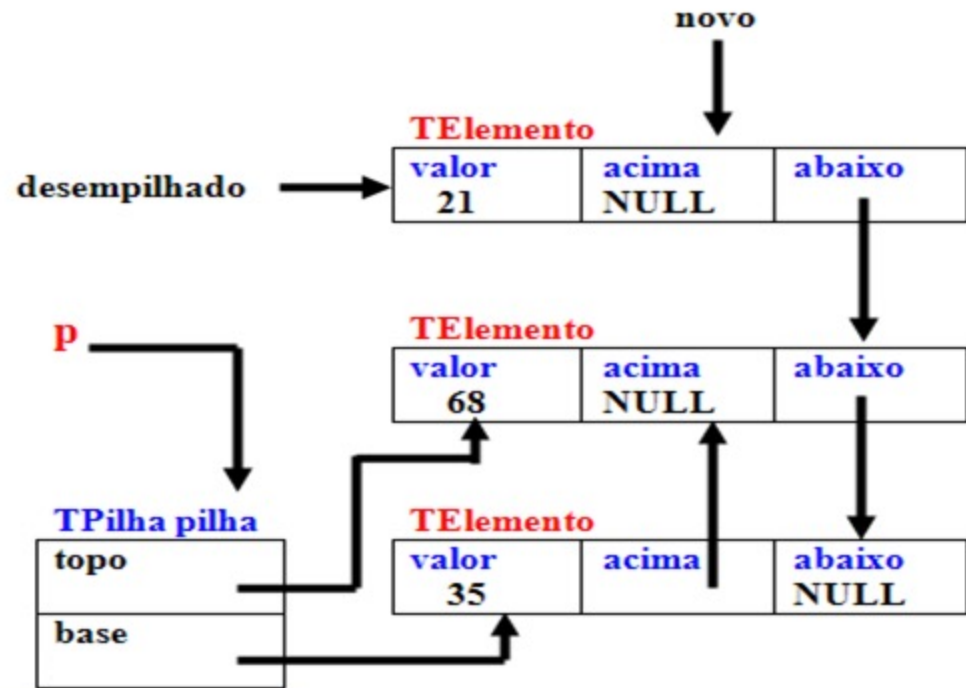
Se a pilha **NÃO**(!) está **VAZIA** então:

o ponteiro **desempilhado** (variável local da função) recebe **novο** (aponta para o mesmo endereço que **novο**).

o ponteiro **topo** de **TPilha** apontado por **p** aponta para o mesmo endereço que o campo **abaixo** apontado por **novο**: o Elemento de valor **68** passa a ser o novo **topo** da **PILHA**.

Entendendo desempilha()

```
if (p->topo == NULL){  
    p->base = NULL;  
} else {  
    p->topo->acima = NULL;  
} //if  
//if  
return desempilhado;  
} //desempilha()  
..
```



Como o **topo** da pilha apontada por **p** **NÃO** é **NULL**: o campo **acima** apontado por **topo** (do **Elemento** de valor **68**) recebe **NULL** (deixa de apontar para o Elemento de valor **21**).

O ponteiro **desempilhado** é retornado pela **função** e ele aponta para o **Elemento** de valor **21**.

Entendendo desempilha()

```
case 2:{
    TElemento *descartado = desempilha(&pilha);
    if (descartado != NULL){
        printf("\n\n\tELEMENTO desempilhado: %d.\n\n", descartado->valor);
        free(descartado);
    } else {
        printf("\n\n\tNENHUM ELEMENTO foi desempilhado, pois a PILHA");
        printf(" encontra-se VAZIA\n\n");
    } //if
    system("PAUSE");
    break;}
}
```

No programa principal (**main()**), o ponteiro (para TElemento) **descartado** recebe da função **desempilha()** o elemento de valor **21**.

Se ponteiro **descartado** **NÃO** for **NULL** (o que é exatamente o caso: ele aponta para **21**):
O conteúdo do campo **valor** apontado por **descartado** é exibido em tela através do **printf()**.

O endereço de memória onde está armazenado o elemento de valor **21** é devolvido ao sistema operacional através da função **free()**.

Para Refletir

- Será que na estrutura de pilha precisamos mesmo de armazenar um ponteiro para a base ou será que conseguiríamos eliminar facilmente esse ponteiro mantendo apenas um ponteiro para o topo???

Exercício Resolvido - 1

- Para resolver uma **expressão matemática** devemos considerar a **prioridade dos operadores**, chamada de **precedência**.
- As **operações** de multiplicação e de divisão têm **prioridade** sobre as **operações** de soma e de subtração.

Exercício Resolvido - 1

- No caso de **operadores** de mesma **prioridade** (**precedência**), os cálculos serão efetuados na ordem em que aparecem na expressão.
- Os **parênteses** podem **alterar** totalmente a **ordem de precedência**.
- Afinal, **$A * B + C$** produz um **resultado diferente** de **$A * (B + C)$** .

Exercício Resolvido - 1

- O matemático polonês **Jan Lukasiewics** elaborou uma saída para **representarmos** e **avaliarmos expressões** sem nos preocuparmos com as **prioridades** das **operações** e até mesmo abrir mão dos **parênteses**.

Exercício Resolvido - 1

Tabela de Conversões de Expressões

Notação <u>infixa</u>	Notação pós-fixa
$A - B * C$	$A B C * -$
$A * (B - C)$	$A B C - *$
$(A - B) / (C + D)$	$A B - C D + /$
$(A - B) / (C + D) * E$	$A B - C D + / E *$
$A + B$	$A B +$
$A - B + C$	$A B - C +$
$A \wedge B * C - D + E / F (G - H)$	$A B \wedge C * D - E F / G H - / +$

Exemplo:

$A - B * C$

Colocar manualmente parênteses na expressão – tornando explícita a precedência das operações que antes estava implícita.

$A - (B * C)$

A multiplicação (*) tem precedência sobre a subtração (-), a não ser que a colocação proposital de parênteses altere a precedência padrão das operações.

Exercício Resolvido - 1

- Percorrer a **expressão** já com **parênteses**, da esquerda para a direita, e para cada **símbolo** (**caractere**) encontrado ao longo da **expressão**, tomar a seguinte decisão:
- Se for **parêntese de abertura**, **ignorá-lo**;
- Se for **operando**, **copiá-lo** para a **expressão pós-fixa** (saída desejada);
- Se for **operador**, **colocá-lo** na **pilha**;
- Se for **parêntese de fechamento**, **desempilhar** o operador presente no topo da **pilha**.
- Ao final deste processo, caso a **pilha não** esteja **vazia**, é um sinal de que **algo de errado ocorreu** durante a **conversão** da notação **infixa** para **pós-fixa**.

Exercício Resolvido - 1

Expressão:

$A - B * C + D$

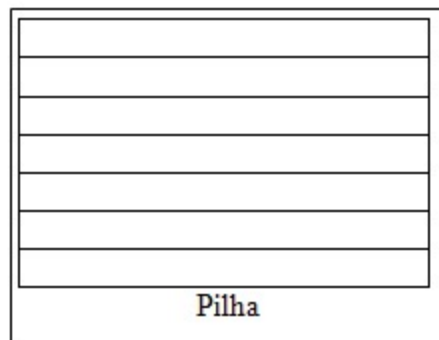


Expressão:

$((A - (B * C)) + D)$

Conversão

Exercício Resolvido - 1

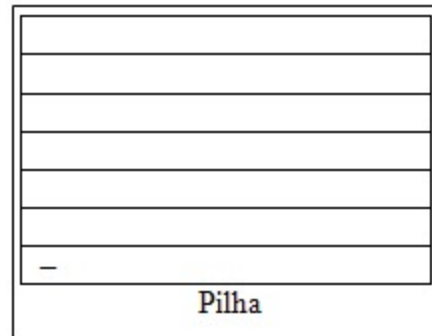


Saída:

A

Expressão:

$((A - (B * C)) + D)$



Saída:

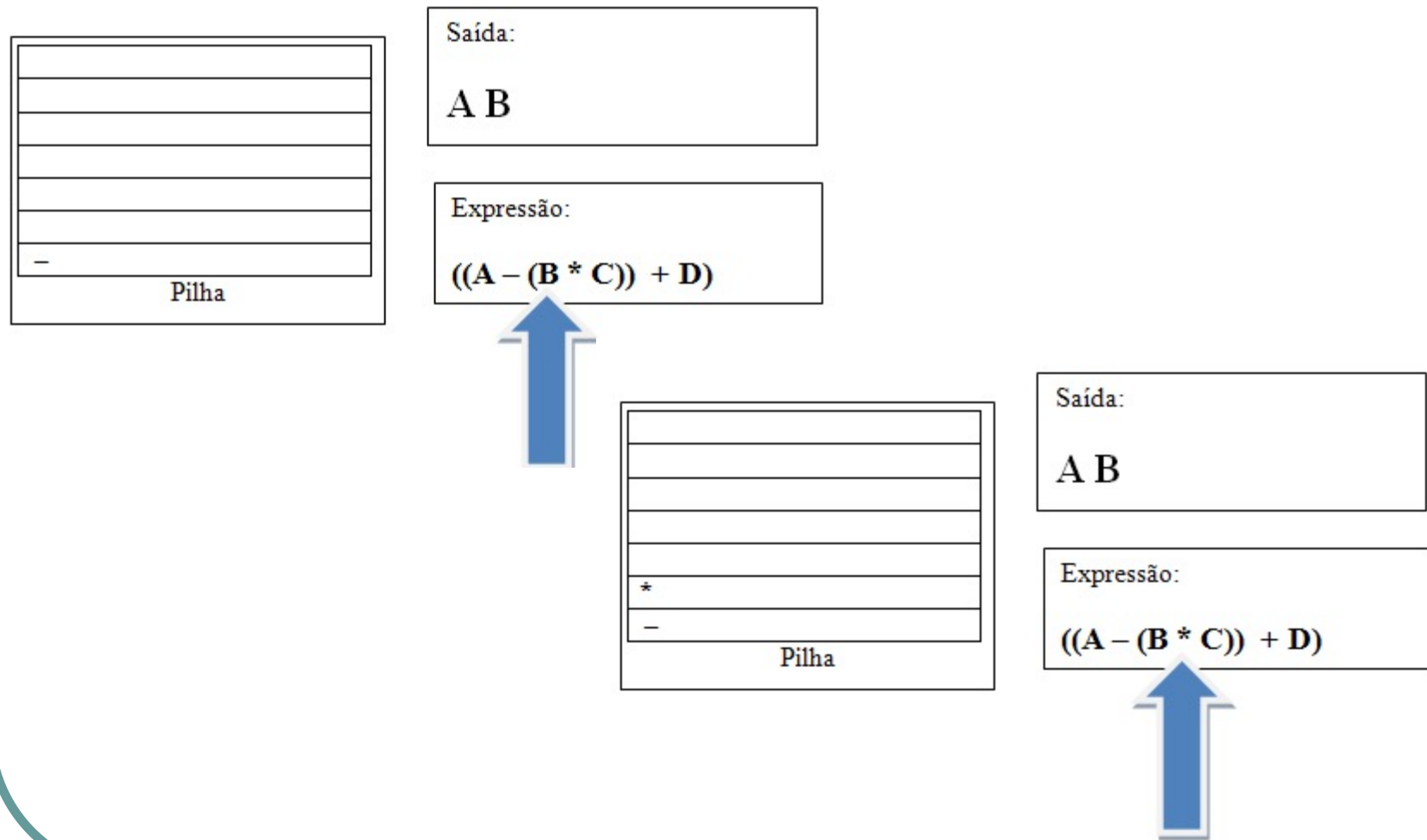
A

Expressão:

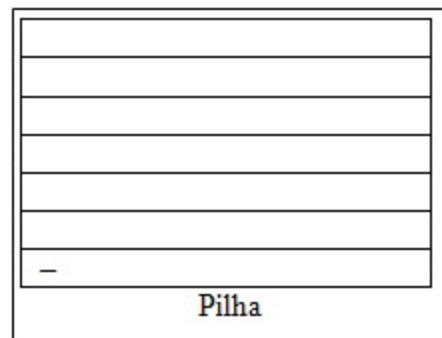
$((A - (B * C)) + D)$



Exercício Resolvido - 1

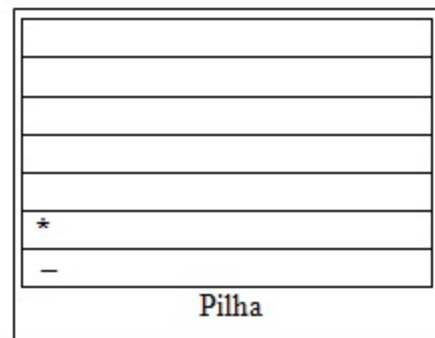



Exercício Resolvido - 1




Saída:
A B C *

Expressão:
 $((A - (B * C)) + D)$

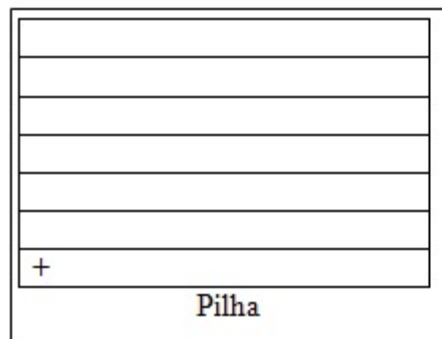


Saída:
A B C

Expressão:
 $((A - (B * C)) + D)$



Exercício Resolvido - 1

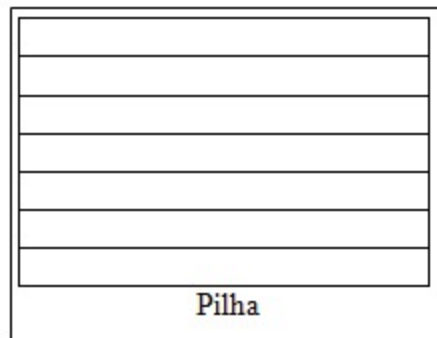



Saída:

A B C * -

Expressão:

$((A - (B * C)) + D)$




Saída:

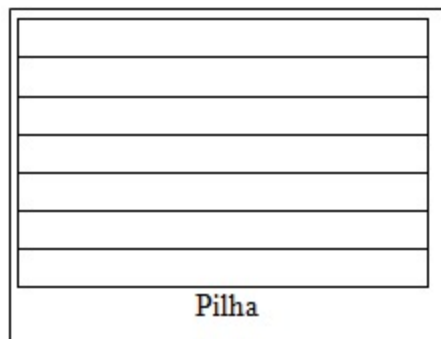
A B C * -

Expressão:

$((A - (B * C)) + D)$



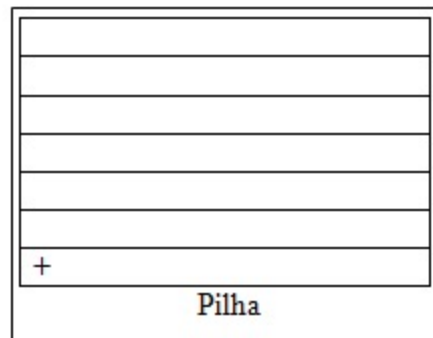
Exercício Resolvido - 1



Saída:

$$A B C^* - D +$$

Expressão:

$$((A - (B * C)) + D)$$


Saída:

A B C * - D

Expressão:

$$((A - (B * C)) + D)$$


Exercício Resolvido - 1

- Implemente esse algoritmo de conversão de uma notação infixa em pós-fixa. Codifique-o na linguagem C.

Exercício Resolvido - 1

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

#define Tamanho 100

typedef struct TipoElemento {
    char valor;
    TipoElemento *acima, *abaixo;
}TElemento;

typedef struct TipoPilha {
    TElemento *topo, *base;
}TPilha;

TPilha pilha;

char expressao[Tamanho];

void formatar(char expr[Tamanho]);
void trataSinal(char expr[Tamanho], char sinal, int inicio);
int localizaSinal(char expr[Tamanho], char sinal, int inicio);
int localizaPreSinal(char expr[Tamanho], int posSinal);
int localizaPosSinal(char expr[Tamanho], int posSinal);
void insereParenteses(char expr[Tamanho], int posPre, int posPos);
void especificacao();
void conversao(char expr[Tamanho], TPilha *pilha);
```

Exercício Resolvido – 1 (Parte 2)

```
void inicPilha(TPilha *p);
int pilhaVazia(TPilha *p);
void empilha(TPilha *p, char caractere);
TElemento *desempilha(TPilha *p);
char elemTopo(TPilha *p);

int main(){
    especificacao();

    printf("\n\n\n\tEscreva EXPRESSAO: ");
    fflush(stdin);
    gets(expressao);

    formatar(expressao);

    printf("\n\n\n\tExpressao FORMATADA: %s\n\n",expressao);
    system("PAUSE");

    conversao(expressao, &pilha);
} //main()
```

Exercício Resolvido – 1 (Parte 3)

```
//=====
void formatar(char expr[Tamanho]){
    trataSinal(expr, '*', 0);
    trataSinal(expr, '/', 0);
    trataSinal(expr, '+', 0);
    trataSinal(expr, '-', 0);
}

//=====
void trataSinal(char expr[Tamanho], char sinal, int inicio){
    int tamanho = strlen(expr);
    int posSinal = localizaSinal(expr, sinal, inicio);
    int posPre = localizaPreSinal(expr, posSinal);
    int posPos = localizaPosSinal(expr, posSinal);

    insereParenteses(expr, posPre, posPos);

    if (posSinal > -1){
        posSinal = posSinal + 2;
        trataSinal(expr, sinal, posSinal);
    }
}

//=====
```

Exercício Resolvido – 1 (Parte 4)

```
//=====
int localizaSinal(char expr[Tamanho], char sinal, int inicio){
    int pos, resultado = -1, tamanho = strlen(expr);
    for (pos = inicio; pos < tamanho; pos++){
        if (expr[pos] == sinal){
            resultado = pos;
            break;
        }
    }
    return resultado;
}

//=====
int localizaPreSinal(char expr[Tamanho], int posSinal){
    int tamanho = strlen(expr);
    int pos = posSinal;
    int resultado = -1;
    int cont = 0;
    if (posSinal > -1){
        for(pos = (posSinal - 1); pos >= 0; pos--){
            if (expr[pos] != ' '){
                if (expr[pos] == ')') cont++;
                if (expr[pos] == '(') cont--;
                if (cont == 0){
                    resultado = pos;
                    break;
                }
            }
        }
    }
    return resultado;
}
```

Exercício Resolvido – 1 (Parte 5)

```
//=====
int localizaPosSinal(char expr[Tamanho], int posSinal){
    int tamanho = strlen(expr);
    int pos = posSinal;
    int resultado = -1;
    int cont = 0;

    if (posSinal > -1){
        for(pos = (posSinal + 1); pos < tamanho; pos++){
            if (expr[pos] != ' '){
                if (expr[pos] == '(') cont++;
                if (expr[pos] == ')') cont--;
                if (cont == 0){
                    resultado = pos;
                    break;
                }
            }
        }
    }
    return resultado;
}
//=====
```

Exercício Resolvido – 1 (Parte 6)

```
//=====
void insereParenteses(char expr[Tamanho], int posPre, int posPos){
    int posLida = 0, posEscr = 0, tamanho = strlen(expr);
    char novaExpr[Tamanho];

    strcpy(novaExpr, "");

    for(posLida = 0; posLida < tamanho; posLida++){
        if (posLida == posPre) novaExpr[posEscr++] = '(';
        novaExpr[posEscr++] = expr[posLida];
        if (posLida == posPos) novaExpr[posEscr++] = ')';
    }//for

    novaExpr[posEscr] = '\0';

    strcpy(expr, novaExpr);
}
//=====
void especificacao(){
    system("CLS");
    printf("\n\nREGRAS para ESCREVER uma EXPRESSAO Valida:\n\n");
    printf("\t(1) - EXPRESSAO NAO deve conter CONSTANTES NUMERICAS: \n");
    printf("\t\tTERRADO:\tA + 3B - 10\t\tCERTO:\tA + B - C\n\n");
    printf("\t(2) - EXPRESSAO NAO deve conter VARIAVEIS com MAIS de uma LETRA no NOME:\n");
    printf("\t\tTERRADO:\tXYZ * AB / RESTO\t\tCERTO:\tX * A / R\n\n");
    system("PAUSE");
    printf("\n\n\t(3) - SOMENTE sao aceitos 4 OPERADORES ARITMETICOS:\n");
    printf("\t\tSOMA: *\n\t\tSUBTRACAO: -\n\t\tMULTIPLICACAO: *\n\t\tDIVISAO: /\n\n");
    printf("\t(4) - PARENTESSES sao PERMITIDOS mas NAO OBRIGATORIOS.\n\n");
    system("PAUSE");
}
```


Exercício Resolvido – 1 (Parte 7)

```
//=====
void inicPilha(TPilha *p){
    p->topo = NULL;
    p->base = NULL;
} //inicPilha()
//.....
int pilhaVazia(TPilha *p){
    if (p->topo == NULL)
        return 1;
    else
        return 0;
} //pilhaVazia()
//.....
```

Exercício Resolvido – 1 (Parte 8)

```
//.....  
void empilha(TPilha *p, char caractere){  
    TElemento *novo = (TElemento *)malloc(sizeof(TElemento));  
    novo->valor = caractere;  
  
    if (pilhaVazia(p)){  
        p->topo = novo;  
        p->base = novo;  
  
        novo->abaixo = NULL;  
    } else {  
        novo->valor = caractere;  
  
        p->topo->acima = novo;  
        novo->abaixo = p->topo;  
        p->topo = novo;  
  
    }  
    novo->acima = NULL;  
  
} //empilha()  
//.....
```

Exercício Resolvido – 1 (Parte 9)

```
TElemento *desempilha(TPilha *p){
    TElemento *desempilhado = NULL;

    if (!pilhaVazia(p)){

        desempilhado = p->topo;

        p->topo = p->topo->abaixo;

        if (p->topo == NULL){
            p->base = NULL;
        } else {
            p->topo->acima = NULL;
        } //if
    } //if
    return desempilhado;
} //desempilha()
//.....
char elemTopo(TPilha *p){
    int valor = -9999;
    if (pilhaVazia(p)){
        printf("\n\nOPERACAO INVALIDA !!!\nPilha encontra-se VAZIA\n");
        printf("Impossivel exibir valor do TOPO da Pilha.\n\n");
        system("PAUSE");
        exit(1);
    } else{
        valor = p->topo->valor;
    } //if
    return valor;
} //elemTopo()
//.....
```

Exercício Resolvido 1(Parte 10)

```
void conversao(char expr[Tamanho], TPilha *pilha){
    int tamanho = strlen(expr);
    int pos, indice = 0;
    char posFixa[Tamanho];
    TElemento *descarte = NULL;

    strcpy(posFixa, "");

    system("CLS");
    printf("\n\n\n\t\t\t====| Conversao |====\n");
    printf("\t\t\t==| da Notacao INFIXA para a POS-FIXA |==\n\n");
    printf("\tEXPRESSAO INFIXA: %s\n\n",expr);

    for(pos = 0; pos <= tamanho; pos++){
        if((expr[pos] == '+') || (expr[pos] == '-') || (expr[pos] == '*') || (expr[pos] == '/')){
            empilha(pilha,expr[pos]);
        } else if(expr[pos] == ' '){
            descarte = desempilha(pilha);
            if (descarte != NULL){
                posFixa[indice++] = descarte->valor;
                posFixa[indice++] = expr[pos];
            } //if
        } else {
            posFixa[indice++] = expr[pos];
        } //if
    } //for
}
```

Exercício Resolvido 1(Parte 11)

```
if (!pilhaVazia(pilha)){  
    printf("\n\n\tPILHA NAO VAZIA !!!\n\tERRO: EXPRESSAO INVALIDA!!!\n\n");  
} else {  
    printf("\n\n\tPILHA VAZIA !!!\n\tEXPRESSAO VALIDA!!!\n\n");  
} //if
```

```
printf("\tEXPRESSAO POS-FIXA: %s\n\n",posFixa);  
system("PAUSE");
```

```
}
```

```
//=====
```