```c
/**
 * Arquivo: stackTAD1.c
 * Versão : 1.0
 * Data   : 2024-10-15 21:53
 * ------------------------
 * Este arquivo implementa a interface stackTAD.h. Nesta implementação o stack
 * terá tamanho fixo máximo. A implementação, em tese, é independente do tipo de
 * dado do elemento armazenado no stack, que foi definido na interface com o
 * nome de "elementoT", mas na prática isso não ocorre nesta implementação: em
 * alguns subprogramas o tratamento de erro depende do tipo "char" (o padrão
 * da interface) para retornar valores apropriados. Isso pode ser melhorado em
 * implementações finais. A estrutura de dados escolhida para armazenar os dados
 * é um array.
 *
 * Baseado em: Programming Abstractions in C, de Eric S. Roberts.
 *             Capítulo 8: Abstract Data Types (pg. 327-347).
 *
 * Prof.: Abrantes Araújo Silva Filho (Computação Raiz)
 *            www.computacaoraiz.com.br
 *            www.youtube.com.br/computacaoraiz
 *            github.com/computacaoraiz
 *            twitter.com/ComputacaoRaiz
 *            www.linkedin.com/company/computacaoraiz
 *            www.abrantes.pro.br
 *            github.com/abrantesasf
 */

/*** Includes: ***/

#include "genlib.h"
#include "math.h"
#include "stackTAD.h"
#include <stdio.h>
#include <stdlib.h>

/*** Constantes Simbólicas: ***/

/**
 * CONSTANTE: TAMMAX
 * -----------------
 * Esta constante especifica o tamanho máximo de espaço a ser alocado para o
 * array que armazenará os elementos do stack. Se o usuário fizer um push de
 * elementos além deste limite, receberá um erro. Se TAMMAX estiver definido
 * como 0 (zero), indica que o array é dinâmico e não tem tamanho máximo
 * limitante.
 */

#define TAMMAX 1000

/*** Tipos de Dados: ***/

/**
 * TIPO: stackTCD
 * --------------
 * O tipo stackTCD é a representação concreta do tipo abstrato de dado stackTAD
 * definido na interface. Nesta implementação os elementos serão armazenados em
 * um array. Como a definição do stackTCD aparece apenas na implementação, e não
 * na interface, podemos alterar esta definição à vontade, desde que a interface
 * não seja alterada e o comportamento do stack seja mantido. A variável inteira
 * "contagem" manterá o número atual de elementos no stack.
```

```c
 */

struct stackTCD
{
    elementoT dados[TAMMAX];
    int contagem;
};

/*** Definições de Subprogramas (comportamentos): ***/

/**
 * FUNÇÃO: criar_stackTAD
 * Uso: stackTAD = criar_stackTAD( );
 * ----------------------------------
 * Aloca memória suficiente para um stackTCD e retorna um ponteiro para esse
 * objeto, através do tipo abstrato stackTAD. Retorna NULL se não for possível
 * alocar a memória.
 */

stackTAD criar_stackTAD (void)
{
    stackTAD S = malloc(sizeof(struct stackTCD));
    if (S == NULL)
    {
        fprintf(stderr, "Erro: não foi possível alocar o stack.\n");
        return NULL;
    }
    S->contagem = 0;
    return S;
}

/**
 * PROCEDIMENTO: remover_stackTAD
 * Uso: remover_stackTAD(&stackTAD);
 * ---------------------------------
 * Ao receber um PONTEIRO para um stackTAD, ou seja, um ponteiro para um
 * ponteiro para stackTCD, libera a memória alocada para o stackTCD e atribui
 * NULL para o ponteiro original (para evitar dangling pointer). Se o
 * ponteiro original já aponta para NULL, não faz nada.
 */

void remover_stackTAD (stackTAD *stack)
{
    if (*stack != NULL)
    {
        free(*stack);
        *stack = NULL;
    }
}

/**
 * PROCEDIMENTO: push
 * Uso: push(stack, elemento);
 * ---------------------------
 * Este procedimento coloca o elemento especificado no topo do stack.
 */

void push (stackTAD stack, elementoT elemento)
{
    if (stack == NULL)
```

```
        fprintf(stderr, "Erro: push em stack null.\n");
    else if (cheia(stack))
        fprintf(stderr, "Erro: o stack está cheio.\n");
    else
        stack->dados[stack->contagem++] = elemento;
}

/**
 * FUNÇÃO: pop
 * Uso: elemento = pop(stack);
 * ---------------------------
 * Retorna o elemento do topo da pilha, ou termina o programa com um erro.
 */

elementoT pop (stackTAD stack)
{
    if (stack == NULL)
    {
        fprintf(stderr, "Erro: pop em stack null.\n");
        exit(1);
    }
    else if (vazia(stack))
    {
        fprintf(stderr, "Erro: stack vazio.\n");
        exit(1);
    }

    return stack->dados[--stack->contagem];
}

/**
 * PREDICADOS: vazia, cheia
 * Uso: if (vazia(stack)) . . .
 *      if (cheia(stack)) . . .
 * ---------------------------
 * Estes predicados retornam TRUE caso a pilha esteja vazia ou cheia,
 * respectivamente. Se o stack for dinâmico, ou seja, sem um tamanho máximo
 * pré-definido, o predicado "cheia" sempre retornará FALSE.
 */

bool vazia (stackTAD stack)
{
    if (stack == NULL)
    {
        printf("Erro: stack null.\n");
        exit(1);
    }
    return (stack->contagem == 0);
}

bool cheia (stackTAD stack)
{
    if (stack == NULL)
    {
        printf("Erro: stack null.\n");
        exit(1);
    }
    return (stack->contagem == TAMMAX);
}
```

```c
/**
 * FUNÇÃO: tamanho
 * Uso: n = tamanho(stack);
 * ------------------------
 * Retorna o tamanho do stack. Situações:
 *
 *     TAMMAX > 0: se tamanho limitado;
 *     TAMMAX = 0: se tamanho ilimitado;
 *     -1        : se ocorrer algum erro.
 */

long int tamanho (stackTAD stack)
{
    if (stack == NULL)
    {
        fprintf(stderr, "Erro: tamanho de stack null.\n");
        return -1;
    }
    return (long int) TAMMAX;
}

/**
 * FUNÇÃO: qtd_elementos
 * Uso: n = qtd_elementos(stack);
 * ------------------------------
 * Esta função retorna a quantidade de elementos atualmente dentro da pilha,
 * ou -1 se ocorrer algum erro.
 */

long int qtd_elementos (stackTAD stack)
{
    if (stack == NULL)
    {
        fprintf(stderr, "Erro: qtd_elementos de stack null.\n");
        return -1;
    }
    return (stack->contagem);
}

/**
 * FUNÇÃO: ver_elemento
 * Uso: elemento = ver_elemento(stack, posicao);
 * ---------------------------------------------
 * Retorna o elemento que está em uma determinada posição do stack, ou termina
 * o programa com um erro.
 */

elementoT ver_elemento (stackTAD stack, int posicao)
{
    if (stack == NULL)
    {
        fprintf(stderr, "Erro: ver_elemento de stack null.\n");
        exit(1);
    }
    else if (vazia(stack))
    {
        fprintf(stderr, "Erro: stack vazio.\n");
        exit(1);
    }
    else if (posicao < 0 || posicao >= stack->contagem)
```

```
    {
        fprintf(stderr, "Erro: posição inválida.\n");
        exit(1);
    }

    return (stack->dados[posicao]);
}

/**
 * FUNÇÃO: espaco_restante
 * Uso: n = espaco_restante(stack);
 * -------------------------------
 * Retorna a quantidade de espaço restante na pilha. Se a pilha é dinâmica,
 * retorna 0; se houver algum erro, retorna -1.
 *
 * TODO: o retorno 0 (zero) pode indicar tanto uma pilha dinâmica sem limite
 *       pré-definido, ou uma pilha cheia. Isso está errado, é preciso ajustar
 *       esse comportamento.
 */

#ifdef debug
long int espaco_restante (stackTAD stack)
{
    if (stack == NULL)
    {
        fprintf(stderr, "Erro: espaco_restante de stack null.\n");
        return -1;
    }
    else if (TAMMAX == 0)
        return 0;

    return (TAMMAX - stack->contagem);
}
#endif

/**
 * PROCEDIMENTO: imprimir_stack
 * Uso: imprimir_stack(stack, limite);
 * -----------------------------------
 * Imprime os elementos da pilha, até um certo limite.
 */

#ifdef debug
void imprimir_stack (stackTAD stack, int limite)
{
    if (stack == NULL)
        fprintf(stderr, "Erro: imprimir_stack de stack null.\n");
    else if (vazia(stack))
        fprintf(stderr, "Erro: stack está vazio.\n");
    else if (limite < 0 || limite > stack->contagem)
        limite = stack->contagem;

    for (int i = 0; i < stack->contagem && i < limite; i++)
        printf("%c\n", stack->dados[i]);
}
#endif
```