



Esercizio: Stack Astratto di Blocchi Dati (C++)

Si richiede di implementare un modulo C++ denominato stack per gestire una struttura dati Stack (Pila) che memorizza puntatori a blocchi di dati allocati dinamicamente.

L'implementazione interna deve utilizzare una lista dinamica doppiamente linkata con puntatori a head e tail.

L'obiettivo principale è garantire l'astrazione dei dati: l'utente del modulo (main.cpp) non deve conoscere la struttura interna della lista o dello stack, interagendo solo tramite tipi opachi e funzioni di interfaccia.

Indicazioni:

1. Strutture Dati e Tipi

Ogni elemento dello Stack deve contenere un puntatore a una struct Data definita come segue:

```
struct Data {  
    char* buffer;      // Puntatore al blocco di dati (stringa letta da file)  
    unsigned long int size; // Dimensione effettiva del buffer in byte  
};
```

All'interno del modulo (stack.cpp), definire le seguenti strutture interne per la gestione della lista: Node, List, e Stack.

Utilizzare le seguenti typedef per creare tipi opachi e maniglie astratte:

```
// stack.h:  
typedef struct Stack *StackHandle; // Maniglia astratta dello Stack
```

```
// stack.cpp:  
// ... definizioni interne di List, Node, Stack  
// typedef struct Node Node; // Per uso interno  
// typedef struct List List; // Per uso interno
```

2. File di Implementazione Richiesti

A. stack.h (Interfaccia Pubblica)

Questo file deve contenere solo le definizioni dei tipi pubblici, le firme delle funzioni e i puntatori opachi. Non deve esporre la definizione delle struct interne (Node, List, Stack).

```
// ===== stack.h =====
```

```
#ifndef STACK_H  
#define STACK_H
```

```
#include <cstdio> // Per FILE* in main, se necessario, o includi in stack.cpp  
#include <stdbool.h>
```

```
// Tipi Pubblici
```

```

struct Data {
    char* buffer;
    unsigned long int size;
};

// Typedef opaco: l'utente vede solo un puntatore alla struct Stack, ma non come è fatta.
typedef struct Stack *StackHandle;

// Puntatore a funzione per il filtraggio
typedef bool (*FilterFunc)(const Data*);

// --- Funzioni di Gestione Base ---
StackHandle createStack();
void destroyStack(StackHandle handle);

// --- Funzionalità Classiche (LIFO) ---
bool push(StackHandle handle, struct Data* data);
struct Data* pop(StackHandle handle);
struct Data* peek(const StackHandle handle);
bool isEmpty(const StackHandle handle);

// --- Funzionalità Avanzate ---
void printStack(const StackHandle handle);
struct Data* search(const StackHandle handle, const Data* target_data);
bool contains(const StackHandle handle, const Data* target_data);
bool stackExtend(StackHandle dest, StackHandle src);
StackHandle subStack(const StackHandle source, FilterFunc filter);

#endif // STACK_H

```

B. stack.cpp (Implementazione Privata)

Questo file deve contenere la definizione delle strutture interne e l'implementazione di tutte le funzioni dichiarate in stack.h.

```
// ===== stack.cpp =====
```

```
#include "stack.h"
#include <iostream>
#include <cstring>
#include <cstdlib>
```

```
// --- Definizioni Interne (Astrazione) ---
```

```
// Nodo della lista doppiamente linkata
typedef struct Node {
    struct Data* data_ptr;
    struct Node* prev;
    struct Node* next;
} Node;
```

```

// Struttura Lista (Contenitore interno)
typedef struct List {
    Node* head; // Cima dello stack
    Node* tail; // Fondo dello stack
    unsigned int count;
} List;

// Struttura Stack (Contiene la lista) - Corrisponde a StackHandle
typedef struct Stack {
    List internal_list;
} Stack;

// -----
// Implementazione: createStack
StackHandle createStack() {
    StackHandle handle = new Stack;
    if (handle) {
        handle->internal_list.head = nullptr;
        handle->internal_list.tail = nullptr;
        handle->internal_list.count = 0;
    }
    return handle;
}

// Implementazione: destroyStack
void destroyStack(StackHandle handle) {
    if (!handle) return;

    // Deallocazione di tutti i nodi e dei loro dati
    Node* current = handle->internal_list.head;
    Node* next;
    while (current != nullptr) {
        next = current->next;
        // 1. Dealloca il buffer char* all'interno della struct Data
        if (current->data_ptr->buffer) {
            delete[] current->data_ptr->buffer;
        }
        // 2. Dealloca la struct Data stessa
        delete current->data_ptr;
        // 3. Dealloca il Nodo
        delete current;
        current = next;
    }

    // Dealloca la struttura Stack (la maniglia)
    delete handle;
}

```

```

// ... Implementazione di tutte le altre funzioni (push, pop, search, subStack, ecc.) ...

// Esempio: Implementazione di push
bool push(StackHandle handle, struct Data* data) {
    if (!handle || !data) return false;

    Node* newNode = new Node;
    if (!newNode) return false;

    newNode->data_ptr = data;
    newNode->prev = nullptr;
    newNode->next = handle->internal_list.head;

    if (handle->internal_list.head != nullptr) {
        handle->internal_list.head->prev = newNode;
    } else {
        handle->internal_list.tail = newNode; // È il primo elemento
    }

    handle->internal_list.head = newNode;
    handle->internal_list.count++;
    return true;
}

// Esempio: Implementazione di pop
struct Data* pop(StackHandle handle) {
    if (!handle || handle->internal_list.head == nullptr) {
        return nullptr;
    }

    Node* temp = handle->internal_list.head;
    struct Data* data_out = temp->data_ptr;

    handle->internal_list.head = temp->next;

    if (handle->internal_list.head != nullptr) {
        handle->internal_list.head->prev = nullptr;
    } else {
        handle->internal_list.tail = nullptr;
    }

    delete temp;
    handle->internal_list.count--;
    return data_out;
}

```

C. main.cpp (Test del Modulo)

Il programma principale deve testare tutte le funzionalità. Si deve simulare la lettura di dati da un file (o da un array di stringhe) e la creazione di uno Stack di Data in cui buffer è allocato dinamicamente per ogni stringa letta.

```
// ===== main.cpp =====
#include "stack.h"
#include <iostream>
#include <cstring>
#include <cstdlib>

// Funzione di filtro di esempio: accetta solo stringhe lunghe (size > 8)
bool filterLongStrings(const struct Data* d) {
    return d->size > 8;
}

// Funzione helper per creare Data* da una stringa
struct Data* createDataFromStr(const char* str) {
    size_t len = std::strlen(str);
    struct Data* d = new struct Data;
    d->size = len;
    d->buffer = new char[len + 1];
    std::strcpy(d->buffer, str);
    return d;
}

int main() {
    std::cout << "--- Test Modulo Stack Astratto ---\n";

    // 1. Creazione dello Stack (Handle opaco)
    StackHandle primaryStack = createStack();

    // Dati di prova (simulano lettura da file)
    const char* file_data[] = {"C++", "Abstraction", "DynamicMemory", "FilterTest", "Short"};

    // 2. Popolamento dello Stack
    for (const char* str : file_data) {
        push(primaryStack, createDataFromStr(str));
    }
    std::cout << "Stack Primario popolato.\n";
    printStack(primaryStack);

    // 3. Test Funzionalità LIFO (Peek e Pop)
    struct Data* peeked = peek(primaryStack);
    if (peeked) {
        std::cout << "\nPeek (Cima): " << peeked->buffer << "\n";
    }

    struct Data* popped = pop(primaryStack);
    if (popped) {
```

```

    std::cout << "Pop (Rimosso): " << popped->buffer << "\n";
    delete[] popped->buffer;
    delete popped;
}

// 4. Test Contains e Search
struct Data target = { (char*)"Abstraction", 13 }; // Usato solo per il confronto del contenuto
std::cout << "\n'Abstraction' e' presente? " << (contains(primaryStack, &target) ? "Si" :
"No") << "\n";

// 5. Test StackExtend (Creazione di un secondo stack)
StackHandle secondaryStack = createStack();
push(secondaryStack, createDataFromStr("Extended Data"));

std::cout << "\nEstensione dello Stack Primario con il Secondario...\n";
stackExtend(primaryStack, secondaryStack);
printStack(primaryStack);

// 6. Test SubStack con Filtro
std::cout << "\nCreazione SubStack (solo stringhe > 8 caratteri)... \n";
StackHandle filteredStack = subStack(primaryStack, filterLongStrings);
printStack(filteredStack);

// 7. Pulizia della memoria
destroyStack(primaryStack);
destroyStack(filteredStack);
// Nota: secondaryStack è stato assorbito da primaryStack con extend, quindi non va
distrutto separatamente.

return 0;
}

```