Programming in Modern C++

Module M05: Stack and Common Data Structures / Containers

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Flexibility of defining *customised* sort algorithms to be passed as parameter to sort and search functions defined in the `algorithm` library
- Predefined optimised versions of these sort and search functions can also be used
- There are a number of useful functions like `rotate`, `replace`, `merge`, `swap`, `remove` etc. in `algorithm` library

Module Objectives

- Understanding implementation and use of stack in C
- Understanding stack in C++ standard library and its use
- Understanding common containers in C++ standard library

Module Outline

Stack in C

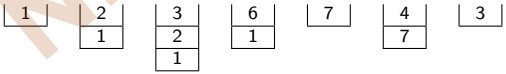
Stack in C

- Stack is a LIFO (last-In-First-Out) container that can maintain a collection of arbitrary number of data items – all of the same type
- To create a stack in C we need to:
  - Decide on the data type of the elements
  - Define a structure (container) (with maximum size) for stack and declare a `top` variable in the structure
  - Write separate functions for `push`, `pop`, `top`, and `isempty` using the declared structure
- **Note**:
  - Change of the data type of elements, implies re-implementation for all the stack codes
  - Change in the structure needs changes in all functions
- Unlike `sin`, `sqrt` etc. function from C standard library, we do not have a ready-made stack that we can use

Common C programs using stack

Some common C programs that use stack:

- *Reversing a string*
  - Input: ABCDE
  - Output: EDCBA
- *Evaluation of postfix expression*
  - Input: 1 2 3 * + 4 - (for 1 + 2 * 3 - 4)
  - Output: 3
  - Stack states:



- *Identification of palindromes* (w/ and w/o center-marker)
- *Conversion of an infix expression to postfix*
- *Depth-first Search* (DFS)

Program 05.01: Reversing a string


```
#include <stdio.h>

typedef struct stack {
    char data [100];
    int top;
} stack;

int empty(stack *p) { return  (p->top  == -1); }

int top(stack *p) { return  p -> data [p->top]; }

void push(stack *p, char x) {
    p -> data [++(p -> top)] = x;
}

void pop(stack *p) {
    if (!empty(p)) (p->top) = (p->top) -1;
}
```

```
int main() {
    stack s;
    s.top = -1;

    char ch, str[10] = "ABCDE";

    int i, len = sizeof(str);

    for(i = 0; i < len; i++)
        push(&s, str[i]);

    printf("Reversed String: ");

    while (!empty(&s)) {
        printf("%c ", top(&s));
        pop(&s);
    }
}
```



```
Reversed String:  EDCBA
```

Program 05.02: Postfix Expression Evaluation

```
#include <stdio.h>

typedef struct stack {
    char data [100];
    int top;
} stack;

int empty(stack *p) {
    return (p->top  == -1);
}
int top(stack *p) {
    return p -> data [p->top];
}
void push(stack *p, char x) {
    p -> data [++(p -> top)] = x;
}
void pop(stack *p) {
    if (!empty(p)) (p->top) = (p->top) -1;
}
```

```
void main() { stack s; s.top = -1;

    // Postfix expression:  1 2 3 * + 4 -
    char postfix[] = {'1','2','3','*','+','4','-'};

    for(int i = 0; i < 7; i++) { char ch = postfix[i];
        if (isdigit(ch)) push(&s, ch-'0');
        else {
            int op2 = top(&s); pop(&s);
            int op1 = top(&s); pop(&s);
            switch (ch) {
                case '+': push(&s, op1 + op2); break;
                case '-': push(&s, op1 - op2); break;
                case '*': push(&s, op1 * op2); break;
                case '/': push(&s, op1 / op2); break;
            }
        }
    }
    printf("Evaluation %d\n", top(&s));
}
```


Evaluation 3

Stack in C++

Understanding Stack in C++

- C++ standard library provide a ready-made stack for any type of elements
- To create a stack in C++ we need to:
  - Include the `stack` header
  - Instantiate a stack with proper element type (like `char`)
  - Use the functions of the stack objects for stack operations

Program 05.03: Reverse a String in C++

```c
#include <stdio.h>
#include <string.h>
#include "stack.h" // User defined codes


int main() { char str[10] = "ABCDE";
    stack s; s.top = -1; // stack struct

    for(int i = 0; i < strlen(str); i++)
        push(&s, str[i]);

    printf("Reversed String: ");
    while (!empty(&s)) {
        printf("%c ", top(&s)); pop(&s);
    }
}
```

- *Lot of code* for creating stack in stack.h
- top to be initialized
- *Cluttered interface* for stack functions
- *Implemented by user* – *error-prone*

```c
#include <iostream>
#include <cstring>
#include <stack> // Library codes
using namespace std;

int main() { char str[10]= "ABCDE";
    stack<char> s; // stack class

    for(int i = 0; i < strlen(str); i++)
        s.push(str[i]);

    cout << "Reversed String: ";
    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
}
```

- *No codes* for creating stack
- *No initialization*
- *Clean interface* for stack functions
- *Available in library* – *well-tested*

Program 05.04: Postfix Evaluation in C++


```
#include <iostream>
#include <stack> // Library codes
using namespace std;

int main() {
    // Postfix expression: 1 2 3 * + 4 -
    char postfix[] = {'1','2','3','*','+','4','-'}, ch;
    stack<int> s; // stack class

    for(int i = 0; i < 7; i++) { ch = postfix[i];
        if (isdigit(ch)) { s.push(ch-'0'); }
        else {
            int op1 = s.top(); s.pop();
            int op2 = s.top(); s.pop();
            switch (ch) {
                case '*': s.push(op2 * op1); break;
                case '/': s.push(op2 / op1); break;
                case '+': s.push(op2 + op1); break;
                case '-': s.push(op2 - op1); break;
            }
        }
    }
    cout << "\nEvaluation " << s.top();
}
```

Data Structures / Containers in C++

Data Structures / Containers in C++

- Like `Stack`, several other data structures are available in C++ standard library
- They are *ready-made* and *work like a data type*
- *Varied types of elements* can be used for C++ data structures
- **Data Structures** in C++ are commonly called **Containers**:
  - A container is a *holder object* that stores a *collection of other objects* (its elements)
  - They are implemented as *class templates* allowing great flexibility in the types supported as elements
  - The container
    - *manages the storage space* for its elements
    - provides member *functions to access* them
    - supports *iterators* - reference objects with similar properties to pointers
  - Many containers have several *member functions in common*, and *share functionalities* - easy to learn and remember
  - `stack`, `queue` and `priority_queue` are implemented as **Container Adaptors**
    - Container adaptors are *not full container classes*, but classes that provide a *specific interface relying on an object of one of the container classes* (such as `deque` or `list`) to handle the elements
    - The underlying container is encapsulated in such a way that its elements are accessed by the members of the container adaptor independently of the underlying container class used

Data Structures / Containers in C++

Container	Class Template	Remarks
Sequence containers: *Elements are* ordered in a strict sequence *and are* accessed by their position in the sequence		
`array` (C++11)	Array class	1D array of *fixed-size*
`vector`	Vector	1D array of *fixed-size* that can *change in size*
`deque`	Double ended queue	*Dynamically sized*, can be expanded / contracted on *both ends*
`forward_list` (C++11)	Forward list	*Const. time insert / erase* anywhere, done as *singly-linked lists*
`list`	List	*Const. time insert / erase* anywhere, iteration in *both directions*
Container adaptors: *Sequence containers* adapted with specific protocols of access *like LIFO, FIFO, Priority*		
`stack`	LIFO stack	Underlying container is `deque` (default) or as specified
`queue`	FIFO queue	Underlying container is `deque` (default) or as specified
`priority_queue`	Priority queue	Underlying container is `vector` (default) or as specified
Associative containers: *Elements are* referenced by their key *and* not by their absolute position *in the container. They are typically implemented as* binary search trees *and needs the elements to be* comparable		
`set`	Set	Stores *unique elements* in *a specific order*
`multiset`	Multiple-key set	Stores elements in *an order* with *multiple equivalent values*
`map`	Map	Stores *<key, value>* in *an order* with *unique keys*
`multimap`	Multiple-key map	Stores *<key, value>* in *an order* with *multiple equivalent values*
Unordered associative containers: *Elements are* referenced by their key *and* not by their absolute position *in the container. Implemented using a* hash table of keys *and has* fast retrieval *of elements based on keys*		
`unordered_set` (C++11)	Unordered Set	Stores *unique elements* in no particular order
`unordered_multiset` (C++11)	Unordered Multiset	Stores elements in *no order* with *multiple equivalent values*
`unordered_map` (C++11)	Unordered Map	Stores *<key, value>* in *no order* with *unique keys*
`unordered_multimap` (C++11)	Unordered Multimap	Stores *<key, value>* in *no order* with *multiple equivalent values*

Module Summary

- In C, stack needs to be coded by the user and works for a specific type of elements only
- C++ standard library provides ready-made `stack`. It works like a data type
- There are several containers in C++ standard library