# Learning C++: The STL and the stack Class

The stack class provides an interface to a very specialized container. A stack is a last-in, first-out (LIFO) container that has a minimum number of member functions you can use to operate on the data stored in a stack. There are several areas of computer science and computer programming where stacks play a prominent role. This article will discuss how to use stacks in your C++ programming and demonstrate some applications where stacks are used.

## A Stack Overview

A stack is a container where all data enters and exits from the top of the container. The three main operations you can perform on a stack are: 1) *push* new data onto the stack; 2) *pop* the top data element off the stack; and 3) view the *top* of the stack.

If you want to think of a real-life corollary, think of the trays at a cafeteria. When a customer enters the cafeteria and gets a tray, they take the tray off the top of the stack. When a dishwasher cleans a tray and puts it back, the tray goes on top of the tray stack. When you look at the tray stack, the only tray you can completely see is the top tray.

As I mentioned earlier, a stack is considered a last-in, first-out container because the last element pushed onto the top of a stack is the first element popped from the top of a stack.

## Declaring Stacks

The header file for the stack class is:

#include <stack>

The stack class is a template class so you must provide a data type with the declaration, as in these examples:

stack<int> numbers;

stack<string> people;

stack<double> ratios;

stack<char> operators;

You can't initialize a stack with an initializer list and you can't specify an initial capacity as a constructor argument as you can with other containers.

## Adding Data to a Stack and Viewing the Top of a Stack

There is only one way to add data to a stack – the push function. The push function takes an element and stores it at the top of the stack. The stack top is also the only element you can access in a stack. The class has a function for examining the top of a stack – the top function.

Here is a short program that pushes some data onto a stack and then examines the top of the stack:

#include <iostream>

#include <stack>

using namespace std;

int main()

{

stack<int> numbers;

numbers.push(2);

numbers.push(4);

numbers.push(8);

cout << "top of stack: " << numbers.top() << endl;

return 0;

}

The output from this program is:

top of stack: 8

## Removing Data from a Stack

Data is removed from a stack with the pop function. This function can only remove the element at the top of the stack. As I mentioned earlier, no other elements on the stack are accessible other than the element at the top.

Here is an example of popping data off a stack, examining the top of the stack after each pop:

int main()

{

stack<int> numbers;

numbers.push(2);

numbers.push(4);

numbers.push(8);

cout << "top of stack: " << numbers.top() << endl;

// top of stack: 8

numbers.pop();

cout << "top of stack: " << numbers.top() << endl;

// top of stack: 4

numbers.pop();

cout << "top of stack: " << numbers.top() << endl;

// top of stack: 2

numbers.pop();

cout << "top of stack: " << numbers.top() << endl;

// this line does not execute

return 0;

}

The last cout statement does not execute because I tried to pop an empty stack. There is a function you can use, empty, that lets you know if a stack is empty or still has data in it. Let's modify the program above to utilize the empty function:

int main()

{

stack<int> numbers;

numbers.push(2);

numbers.push(4);

numbers.push(8);

while (!numbers.empty()) {

cout << "top of stack: " << numbers.top() << endl;

numbers.pop();

}

return 0;

}

Another function you can use to help determine when a stack is empty is the size function. This function returns the number of elements in a stack. Here's how you can use it to keep from popping an empty stack:

int main()

{

stack<int> numbers;

numbers.push(2);

numbers.push(4);

numbers.push(8);

while (numbers.size() > 0) {

cout << "top of stack: " << numbers.top() << endl;

numbers.pop();

}

return 0;

}

## Some Stack Applications

Stacks can be used in the service of several interesting algorithms and applications. Here are a few of them.

### Converting Decimal Numbers to Binary

The first algorithm I'll discuss is one for converting a base 10 (decimal) number to a base 2 (binary) number. Here is the algorithm w here n is the decimal number and B is the base (2):

1. *Start with an empty stack.*
2. *Get the right-most digit of n by n % B. Push the result on the stack.*
3. *Replace n with n/B.*
4. *Repeat steps 1 and 2 until n = 0.*
5. *Pop and print the digits from the stack, resulting in the binary number.*

Here is the program that implements this algorithm:

int main()

{

stack<int> binDigits;

int decNumber;

cout << "Enter a decimal number: ";

cin >> decNumber;

int number = decNumber;

const int BASE = 2;

int digit;

while (decNumber > 0) {

digit = decNumber % BASE;

binDigits.push(digit);

decNumber /= BASE;

}

string binNumber = "";

while (!binDigits.empty()) {

binNumber += to\_string(binDigits.top());

binDigits.pop();

}

cout << number << " is " << binNumber

<< " in binary." << endl;

return 0;

}

Here is one run of this program:

Enter a decimal number: 53

53 is 110101 in binary.

### Finding Palindromes

A word is a palindrome if it can be spelled the same forwards and backwards. The words radar and racecar are examples of palindromes. We can use a stack to determine if a word is a palindrome by pushing each letter of a word onto a stack. Then, we can form the reverse of the word by popping the stack into another word until the stack is empty. If the original word and the word formed from the stack are equal, then the word is a palindrome.

Here is a program that checks to see if a word is a palindrome:

int main()

{

stack<char> letters;

string word;

cout << "Enter a word to check: ";

cin >> word;

for (unsigned i = 0; i < word.size(); i++) {

letters.push(word[i]);

}

string revWord = "";

while (!letters.empty()) {

revWord += letters.top();

letters.pop();

}

if (revWord == word) {

cout << word << " is a palindrome." << endl;

}

else {

cout << word << " is not a palindrome." << endl;

}

return 0;

}

Here are a few runs of the program:

Enter a word to check: radar

radar is a palindrome.

Enter a word to check: racecar

racecar is a palindrome.

Enter a word to check: railcar

railcar is not a palindrome.

### Converting an Infix Expression to Postfix

The arithmetic statements we are accustomed to writing are written in infix, such as: (9.0/5.0) \* 100 + 32.0. However, we must use parentheses to indicate the proper order of operations if we can't rely directly on the normal order of operations. Postfix expressions, on the other hand, can be written without parentheses and the proper order of operations is maintained.

Let's build a simple infix-to-postfix converter that works for arithmetic statements that don't include parentheses, such as *a+b\*c*. For this converter, we will push arithmetic operators onto the stack based on their precedence and put identifiers onto the postfix statement.

Here is the program:

#include <iostream>

#include <stack>

using namespace std;

int getPrecedence(char op) {

if (op == '+' || op == '-') {

return 1;

}

else if (op == '\*' || op == '/') {

return 2;

}

else if (op == '^') {

return 3;

}

else {

return 0;

}

}

bool isOp(char ch) {

if (ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == '^') {

return true;

}

return false;

}

int main()

{

stack<char> s;

string infix = "a+b\*c";

string postfix = "";

char ch;

for (unsigned i = 0; i < infix.size(); i++) {

ch = infix[i];

if (isOp(ch)) {

if (s.empty()) {

s.push(ch);

}

else if (getPrecedence(ch) >= getPrecedence(s.top())) {

s.push(ch);

}

}

else {

postfix += ch;

}

}

if (!s.empty()) {

while (!s.empty()) {

postfix += s.top();

s.pop();

}

}

cout << infix << endl;

cout << postfix << endl;

return 0;

}

A great extension to this program would be to add parentheses checking so you can convert even more complex expressions. If you make this extension, I would love to see your program.

## Using Stacks

I have just shown you three examples of using stacks to solve programming problems. In more technical computer science, stacks are used for managing functions (see [call stack](https://en.wikipedia.org/wiki/Call_stack)), managing memory (see [stack-based memory allocation](https://en.wikipedia.org/wiki/Stack-based_memory_allocation)), and in algorithms for backtracking (see [backtracking](https://en.wikipedia.org/wiki/Backtracking)) and to make many other algorithms more efficient.

Thanks for reading this article and please email with comments and suggestions.