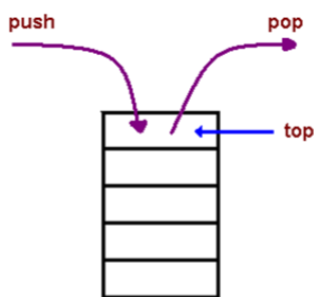
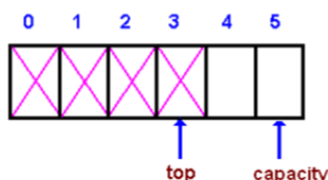
Abstract
A stack is a linear data structure for collection of items, with the restriction that items can be added one at a time and can only be removed in the reverse order in which they were added. The last item represents the top of the stack. Such a stack resembles a stack of trays in a cafeteria, or stack of boxes. Only the top tray can be removed from the stack and it is the last one that was added to the stack. A tray can be removed only if there are some trays on the stack, and a tray can be added only if there is enough room to hold more trays.

Introduction
A Stack is a restricted ordered sequence in which we can only add to and remove from one end — the top of the stack. Imagine stacking a set of books on top of each other — you can push a new book on top of the stack, and you can pop the book that is currently on the top of the stack. You are not, strictly speaking, allowed to add to the middle of the stack, nor are you allowed to remove a book from the middle of the stack. The only book that can be taken out of the stack is the most recently added one; a stack is thus a "last in, first out" (LIFO) data structure.
We use stacks everyday — from finding our way out of a maze, to evaluating postfix expressions, "undoing" the edits in a word-processor, and to implementing recursion in programming language runtime environments.

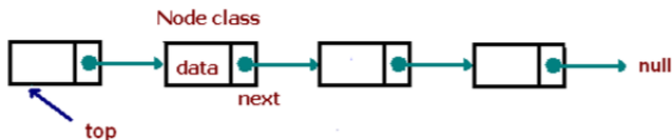
Five basic stack operations are:
1. **push**: adds a new item on top of a stack.
2. **pop**: removes the item on the top of a stack
3. **isEmpty**: Check to see if the stack is empty
4. **isFull**: Check to see if stack is already full
5. **returnTop**: Indicate which item is at the top



Implementation of Stack
A stack is commonly and easily implemented using either an array or a linked list. In the latter case, the head points to the top of the stack: so addition/removal (push/pop) occurs at the head of the linked list.
Array based implementation
In an array-based implementation we add the new item being pushed at the end of the array, and consequently pop the item from the end of the array as well. This way, there is no need to shift the array elements. The top of the stack is always the last used slot in the array, so we can use size-1 to refer to the top of the stack element instead of having to keep a separate field. The top of the stack is not defined for an empty stack.



Linked list based Implementation
Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation.



It shows a partial head-referenced singly-linked based implementation of an unbounded stack. In a singly-linked list-based implementation we add the new item being pushed at the beginning of the array, and consequently pop the item from the beginning of the list as well.
In a bounded or fixed-size stack abstraction, the capacity stays unchanged, therefore when top reaches capacity, the stack object throws an exception.
In an unbounded or dynamic stack abstraction when top reaches capacity, we double up the stack size. The following shows a partial array-based implementation of an unbounded stack.

A stack program: Below is essentially the same stack implemented in C, C++, and Java. This is an *array* implementation. The code is arranged so that similar lines of each language are in a horizontal row with one another. To better illustrate the principles, this stack is ultimately stripped down, without even error checking. Each of the implementations takes place in a separate file, and each implementation is more-or-less how one might write code in the given language. Commentary about these approaches come after the code.
Even the C implementation achieves a key goal of an *abstract data type*: the method of implementation of the stack could be completely changed, say to a linked list form, without altering or even recompiling files that contain code making use of the stack. The C++ and Java stacks have this property also.

C Stack	**C++ Stack**	**Java Stack**
/* stack.h: header file */	// stack.h: header file	// Stack.java: stack
void push(char);	class Stack {	implementation
char pop();	intMaxStack;	public class Stack {
int empty();	intEmptyStack;	private intmaxStack;
int full();	int top;	private intemptyStack;
----------------------------	char* items;	private int top;
	public:	private char[] items;
/* stack.c: stack implem */	Stack(int);	
#include "stack.h"	~Stack();	
	void push(char);	
	char pop();	
#define MAXSTACK 10	int empty();	
#define EMPTYSTACK -	int full();	
1	};	
int top = EMPTYSTACK;	------------------------------	
char items[MAXSTACK];	// stack.cpp: stack functions	
	#include "stack.h"	
	Stack::Stack(int size) {	
	MaxStack = size;	public Stack(int size) {
void push(char c) {	EmptyStack = -1;	maxStack= size;
items[++top] = c;	top = EmptyStack;	emptyStack = -1;

```
}

char pop() {
  return items[top--];
}

int full()  {
  return top+1 ==
MAXSTACK;
}

int empty()  {
  return top ==
EMPTYSTACK;
}

----------------------------
/* stackmain.c: use stack
*/
#include <stdio.h>
#include "stack.h"

int main() {


  char ch;
  while ((ch = getchar())
      != '\n')
    if (!full()) push(ch);
  while (!empty())
printf("%c", pop());
printf("\n");
  return 0;
}
```

```
  items = new
char[MaxStack];
}

Stack::~Stack() {delete[]
items;}

void Stack::push(char c) {
  items[++top] = c;
}

char Stack::pop() {
  return items[top--];
}

int Stack::full()  {
  return top + 1 == MaxStack;
}

int Stack::empty()  {
  return top == EmptyStack;
}

------------------------------
// stackmain.cpp: use stack
#include <iostream.h>
#include "stack.h"

int main() {

  Stack s(10); // 10 chars
  char ch;
  while ((ch = cin.get())
      != '\n')
    if (!s.full()) s.push(ch);
  while (!s.empty())
cout<<s.pop();
cout<<endl;
  return 0;
}
```

```
    top = emptyStack;
    items = new char[maxStack];
  }


  public void push(char c) {
    items[++top] = c;
  }

  public char pop() {
    return items[top--];
  }

  public boolean full()  {
    return top + 1 == maxStack;
  }

  public boolean empty()  {
    return top == emptyStack;
  }
}
-------------------------------------
// Stackmain.java: use the stack
import java.io.*;
public class Stackmain {

 public static void main(String[]
args)
     throws IOException {
   Stack s = new Stack(10); // 10
chars
   char ch;
   while ((ch =
(char)System.in.read())
        != '\n')
      if (!s.full()) s.push(ch);
   while (!s.empty())
System.out.print(s.pop());
System.out.println();
  }
}
```

Execution of each program

% **cc -o stack stack.cstackmain.c**

% **CC -o stack stack.cpp stackmain.cpp**

% **javac Stack.java**
% **javac Stackmain.java**

% **stack mississippi** ppississim	stack.cpp: stackmain.cpp: % **stack mississippi** ppississim	% **java Stackmain mississippi** ppississim

Overview of the C Implementation: This is the standard way to "fake" object-oriented programming in C: Use a separate file for the object, in this case **stack.c**. This allows for hidden data fields and hidden "member functions", as if they were declared private in the class of an object-oriented language. Data in the separate file is not accessible to the outside unless the data is declared **extern**. Functions can be protected from outside access by declaring them **static**, a very strange designator that should be called "private". The common connections between the stack implementation and its use occur in a header file, which is included below in each of the other files. This file is **stack.h** below. In this case the header file gives prototypes for the four functions that provide access to the stack. All variables related to the stack implementation are hidden in the implementation file. The header file allows the stack implementation file and any file using the stack to be separately compiled. This C implementation gives a *single instance* of a stack of **char**s of size 10. If one wants a stack of a different size, or of a different type, one must hack the code and recompile. If one wants several instances of this stack (or similar ones), one must make another copy of the file with separate external function names.

Overview of the C++ Implementation: The C++ implementation achieves the goals of inaccessible data and only the desired stack management functions publicly accessible. Even this stripped-down C++ implementation gives far more than the C version: Here a program that uses the stack can instantiate as many stacks of different sizes as desired, and use them all simultaneously.
This same stack code could be written with C "templates" in such a way that a stacks of any type, including user-defined types, could be easily created. For a program that converts this stack code to a template version, see C++ Stack Template
The C++ implementation uses a separate "header" file as shown, similar to the C header file. This gives an "overview" of the stack without the complete implementation, and allows separate compilation of the file to implement the stack and any file that uses the stack. As in C, the header file allows the stack implementation file and any file using the stack to be separately compiled.
This C++ code needs a separate destructor to recover the storage for the stack after it is no longer in use. This is because the one portion of storage is a dynamically allocated array which would not be deleted by the default destructor. Java uses automatic garbage collection, and so it doesn't need destructors.

Overview of the Java Implementation: Many of the same comments given above for C++ apply to the Java implementation. Java doesn't have the C++ templates, but in Java one can create a stack of any type of object, and thus get the same features described above for C++, though the code will not be as efficient.

The Java implementation does not have a separate header file. In order to compile a file that uses this stack, the compiler must have access to the stack implementation file.
Java has the concept of an *interface*, which for a stack such as this would be a contract to implement the four given stack functions, by providing their prototypes in an interface declaration, and saying that the stack "implements" this interface.

Hardware Implementations of Stack

Hardware implementation of stacks has the obvious advantage that it can be much faster than software management. In machines that refer to the stack with a large percentage of instructions, this increased efficiency is vital to maintaining high system performance.
While any software method of handling stacks can be implemented in hardware, the generally practiced hardware implementation is to reserve contiguous locations of memory with a stack pointer into that memory. Usually the pointer is a dedicated hardware register that can be incremented or decremented as required to push and pop elements. Sometimes a capability is provided to add an offset to the stack pointer to non-destructively access the first few elements of the stack without requiring successive pop operations. Often times the stack is resident in the same memory devices as the program. Sometimes, in the interest of increased efficiency, the stacks reside in their own memory devices.
Another approach that may be taken to building stacks in hardware is to use large shift registers. Each shift register is a long chain of registers with one end of the chain being visible as a single bit at the top of stack. 32 such shift registers of N bits each may be placed side-by-side to form a 32 bit wide by N element stack. While this approach has not been practical in the past, VLSI stack machines may find this a viable alternative to the conventional register pointing into memory implementation.

Single vs. Multiple stacks

The most obvious example of a stack supported function is a single stack used to support subroutine return addresses. Often times this stack also is used to pass parameters to subroutines. Sometimes one or more additional stacks are added to allow processing subroutine calls without affecting parameter lists, or to allow processing values on an expression stack separately from subroutine information.
Single Stack computers are those computers with exactly one stack supported by the instruction set. This stack is often intended for state saving for subroutine calls and interrupts. It may also be used for expression evaluation. In either case, it is probably used for subroutine parameter passing by compilers for some languages. In general, a single stack leads to simple hardware, but at the expense of intermingling data parameters with return address information.
An advantage of having a single stack is that it is easier for an operating system to manage only one block of variable sized memory per process. Machines built for structured programming languages often employ a single stack that combines subroutine parameters and the subroutine return address, often using some sort of frame pointer mechanism.
A disadvantage of a single stack is that parameter and return address information are forced to become mutually well nested. This imposes an overhead if modular software design techniques force elements of a parameter list to be propagated through multiple layers of software interfaces, repeatedly being copied into new activation records.

Multiple Stack computers have two or more stacks supported by the instruction set. One stack is usually intended to store return addresses, the other stack is for expression evaluation and subroutine parameter passing. Multiple stacks allow separating control flow information from data operands.
In the case where the parameter stack is separate from the return address stack, software may pass a set of parameters through several layers of subroutines with no overhead for recopying the data into new parameter lists.
An important advantage of having multiple stacks is one of speed. Multiple stacks allow access to multiple values within a clock cycle. As an example, a machine that has simultaneous access to both a data stack and a return address stack can perform subroutine calls and returns in parallel with data operations.

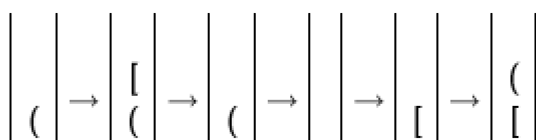
Function Calls: We have already seen the role stacks plays in nested function calls. When the main program calls a function named F, a stack frame for F gets pushed on top of the stack frame for main. If F calls another function G, a new stack frame for G is pushed on top of the frame for F. When G finishes its processing and returns, its frame gets popped off the stack, restoring F to the top of the stack.
Large number Arithmetic: As another example, consider adding very large numbers. Suppose we wanted to add 353,120,457,764,910,452,008,700 and 234,765,000,129,654,080,277. First of all note that it would be difficult to represent the numbers as integer variables, as they cannot hold such large values. The problem can be solved by treating the numbers as strings of numerals, store them on two stacks, and then perform addition by popping numbers from the stacks.

Application of Stacks

- **Balanced Brackets**

A common application of stacks is the parsing and evaluation of arithmetic expressions. Indeed, compilers use a stack in parsing (checking the syntax of) programs. Consider just the problem of checking the brackets/parentheses in an expression. Say [(3+4)*(5–7)]/ (8/4). The brackets here are okay: for each left bracket there is a matching right bracket. Actually, they match in a specific way: two pairs of matched brackets must either nest or be disjoint. You can have [()] or [] (), but not ([)] we can use a stack to store the unmatched brackets. The algorithm is as follows:

Scan the string from left to right, and for each char: 1. If a left bracket, push onto stack 2. If a right bracket, pop bracket from stack (if not match or stack empty then fail) at end of string, if stack empty and always matched, then accept.
For example, suppose the input is: ([]) [(]) Then the stack goes:



- **Reverse**

The simplest application of a stack is to reverse a word. You push a given word to stack – letter by letter – and then pop letters from the stack. Here's the trivial algorithm to print a word in reverse:
begin with an empty stack and an input stream.
while there is more characters to read, do:
read the next input character;
push it onto the stack;
end while;
while the stack is not empty, do:
c = pop the stack;
print c;
end while;

- **Undo**

Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack. Popping the stack is equivalent to "undoing" the last action. A very similar one is going back pages in a browser using the back button; this is accomplished by keeping the pages visited in a stack. Clicking on the back button is equivalent to going back to the most-recently visited page prior to the current.

- **Expression evaluation:**

When an arithmetic expression is presented in the postfix form, you can use a stack to evaluate it to get the final value. For example: the expression 3 + 5 * 9(which is in the usual infix form) can be written as 3 5 9 * + in the postfix. More interestingly, postfix form removes all parentheses and thus all implicit precedence rules; for example, the infix expression ((3 + 2) * 4) / (5 - 1) is written as the postfix 3 2 + 4 * 5 1 - /. You can now design a calculator for expressions in postfix form using a stack.
The algorithm may be written as the following:
Let's apply this algorithm to evaluate the postfix expression 3 2 + 4 * 5 1 - / using a stack.

```
begin with an empty stack and an input stream (for the expression).
while there is more input to read, do:
   read the next input symbol;
   if it's an operand,
      then push it onto the stack;
   if it's an operator
      then pop two operands from the stack;
           perform the operation on the operands;
           push the result;
end while;
// the answer of the expression is waiting for you in the stack:
pop the answer;
```

```
 Stack    Expression

|   |
 ---       3 2 + 4 * 5 1 - /
(empty)

| 3 |      2 + 4 * 5 1 - /
 ---

| 2 |
| 3 |      + 4 * 5 1 - /
 ---

| 5 |      4 * 5 1 - /
 ---

| 4 |
| 5 |      * 5 1 - /
 ---

| 20|      5 1 - /
 ---

| 5 |
| 20|      1 - /
 ---

| 1 |
| 5 |
| 20|      - /
 ---

| 4 |
| 20|      /

| 5 |     (finished. the result is on top of the stack)
 ---
```

- **Backtracking**

This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?

Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

- **Converting an Infix Expression to Postfix**

A stack can also be used to convert an infix expression in standard form into postfix form. We shall assume that the expression is a legal one (i.e. it is possible to evaluate it). When an operand is read, it will be placed on output list (printed out straight away). The operators are pushed on a stack. However, if the priority of the top operator in the stack is higher than the operator being read, then it will be put on output list, and the new operator pushed on to the stack. The priority is assigned as follows.

1. (Left parenthesis in the expression
2. * /
3. + –
4. (Left parenthesis inside the stack

The left parenthesis has the highest priority when it is read from the expression, but once it is on the stack, it assumes the lowest priority.

To start with, the stack is empty. The infix expression is read from left to right. If the character is an OPERAND, it is not put on the stack. It is simply printed out as part of the post fix expression. The stack stores only the OPERATORS. The first operator is pushed on the stack. For all subsequent operators, priority of the incoming operator will be compared with the priority of the operator at the top of the stack.
If the priority of the incoming-operator is higher than the priority of topmost operator-on-the-stack, it will be pushed on the stack. If the priority of the incoming-operator is same or lower than the priority of the operator at the top of the stack, then the operator at top of the stack will be popped and printed on the output expression.
The process is repeated if the priority of the incoming-operator is still same or lower than the next operator-in-the stack. When a left parenthesis is encountered in the expression it is immediately pushed on the stack, as it has the highest priority. However, once it is inside the stack, all other operators are pushed on top of it, as its inside-stack priority is lowest.
When a right parenthesis is encountered, all operators up to the left parenthesis are popped from the stack and printed out. The left and right parentheses will be discarded. When all characters from the input infix expression have been read, the operators remaining inside the stack, are printed out in the order in which they are popped.

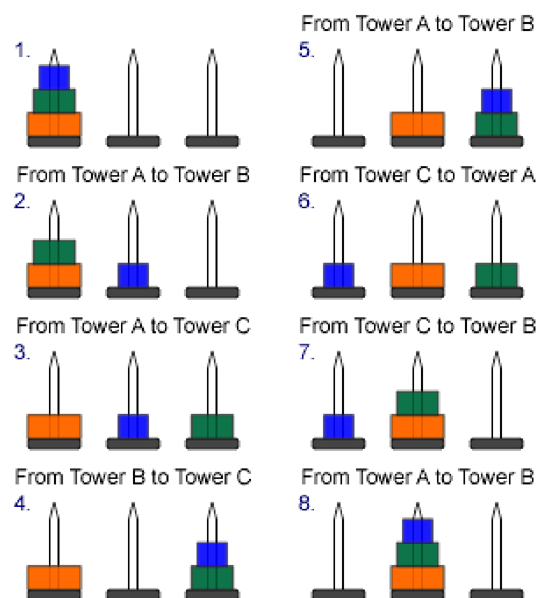
- **Towers of Hanoi**

One of the most interesting applications of stacks can be found in solving a puzzle called Tower of Hanoi. According to an old Brahmin story, the existence of the universe is calculated in terms of the time taken by a number of monks, who are working all the time, to move 64 disks from one pole to another. But there are some rules about how this should be done, which are:

- move only one disk at a time.
- for temporary storage, a third pole may be used.
- a disk of larger diameter may not be placed on a disk of smaller diameter.[8]

For algorithm of this puzzle see Tower of Hanoi.

- Assume that A is first tower, B is second tower & C is third tower.



- **Security**

Some computing environments use stacks in ways that may make them vulnerable to security breaches and attacks. Programmers working in such environments must take special care to avoid the pitfalls of these implementations.
For example, some programming languages use a common stack to store both data local to a called procedure and the linking information that allows the procedure to return to its caller. This means that the program moves data into and out of the same stack that contains critical

return addresses for the procedure calls. If data is moved to the wrong location on the stack, or an oversized data item is moved to a stack location that is not large enough to contain it, return information for procedure calls may be corrupted, causing the program to fail. Malicious parties may attempt a stack smashing attack that takes advantage of this type of implementation by providing oversized data input to a program that does not check the length of input. Such a program may copy the data in its entirety to a location on the stack, and in so doing it may change the return addresses for procedures that have called it. An attacker can experiment to find a specific type of data that can be provided to such a program such that the return address of the current procedure is reset to point to an area within the stack itself (and within the data provided by the attacker), which in turn contains instructions that carry out unauthorized operations.

This type of attack is a variation on the buffer overflow attack and is an extremely frequent source of security breaches in software, mainly because some of the most popular compilers use a shared stack for both data and procedure calls, and do not verify the length of data items.

Future scope of stacks

- **Virtual memory and memory protection**

The use of virtual memory and memory protection are concepts that have not yet been widely incorporated into existing stack machines. This is because most stack machine applications to date have been relatively small programs with tight constraints on hardware and software that did not require or leave room for these techniques.

Memory protection is sometimes important

Memory management can mean many things, but in this case we will focus only on memory management as it pertains to protection features. Protection is the one feature of memory management that is seen as most important by some real time control users, especially the military.

Virtual memory is not used in controllers

Likewise, there is no reason why stack machines cannot be provided with virtual memory capabilities. The one problem with virtual memory is the effect of a virtual memory miss, which may require retrying an instruction. Since stack machines are in essence load/store machines, instruction restart ability is no harder than on a RISC machine.

- **The use of a third stack**

An often proposed design alternative for stack machines is the use of a third hardware stack. The purposes given for adding a third hardware stack are usually for storage of loop counters and local variables.

Loop counters on the current stack machines are generally kept as the top element of the return address stack. This is because subroutines and loops are mutually well nested, and it is considered bad programming style for a subroutine to attempt to access the loop index of its parent procedure.

References

SamiranChattopadhyay,Debabrata Ghosh Dastidar,MatanginiChattopadhyay, Data Structures Through 'C' language Page No. 114-140

http://en.wikipedia.org/wiki/Stack_%28abstract_data_type%29

http://en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queues

http://www.cplusplus.com/reference/stack/stack/

http://pages.cs.wisc.edu/~siff/CS367/Notes/stacks.html

http://www.cs.cmu.edu/~fp/courses/15122-f10/lectures/08-stacks.pdf

Philip Koopman,Jr. , Stack Computers New Wave Page No. 24-27

Philip Koopman,Jr. , Stack Computers New Wave Page No. 175-176