Stacks

Paper: Data Structures
Unit No : II, Linear Structures,
Chapter Name: Stacks

Author: Vandana Kalra, Associate Professor
College/Department: College of Vocational Studies , University of Delhi.

Table of Contents

2 Stacks

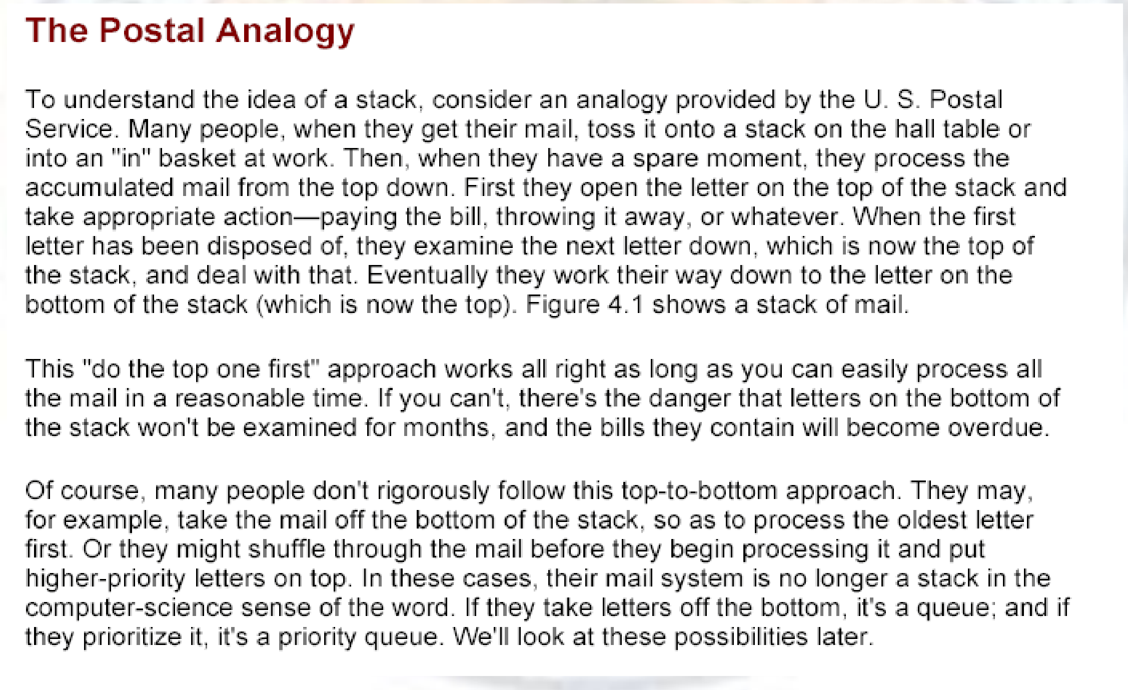
Stack is an abstract data type which operates on data in the specific manner. Implementation of the data types can be done after analyzing the operations required on data efficiently.

2.1 Introduction

At a logical level, Stack is an ordered list of homogeneous elements or items. Addition of new elements and removal of existing elements can takes place from one end of the stack. It can be properly defined as *"It is a Linear data structure containing data items of similar type in which the elements are added and removed from only one end called top of the stack .It works on a principle of "LIFO" ; " Last In First Out ".*

Value addition: Analogy

Data structure Stack



The Postal Analogy

To understand the idea of a stack, consider an analogy provided by the U. S. Postal Service. Many people, when they get their mail, toss it onto a stack on the hall table or into an "in" basket at work. Then, when they have a spare moment, they process the accumulated mail from the top down. First they open the letter on the top of the stack and take appropriate action—paying the bill, throwing it away, or whatever. When the first letter has been disposed of, they examine the next letter down, which is now the top of the stack, and deal with that. Eventually they work their way down to the letter on the bottom of the stack (which is now the top). Figure 4.1 shows a stack of mail.

This "do the top one first" approach works all right as long as you can easily process all the mail in a reasonable time. If you can't, there's the danger that letters on the bottom of the stack won't be examined for months, and the bills they contain will become overdue.

Of course, many people don't rigorously follow this top-to-bottom approach. They may, for example, take the mail off the bottom of the stack, so as to process the oldest letter first. Or they might shuffle through the mail before they begin processing it and put higher-priority letters on top. In these cases, their mail system is no longer a stack in the computer-science sense of the word. If they take letters off the bottom, it's a queue; and if they prioritize it, it's a priority queue. We'll look at these possibilities later.

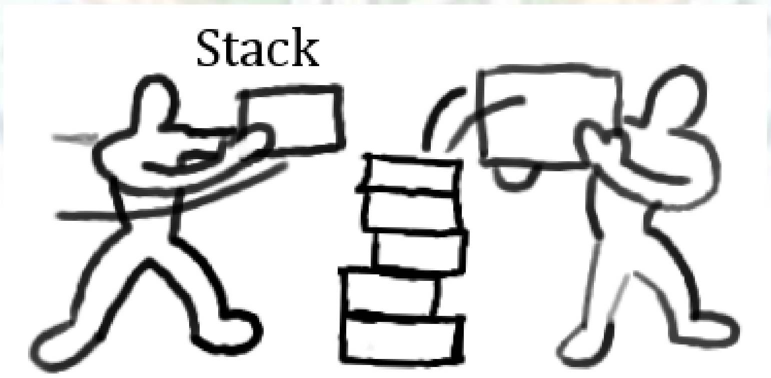
Source: Data Structure And Algorithms In Java - Mitchel Waite

Stacks

In real life application, suppose we have a pile of books arranged one upon the other. Now to remove the second last book we have to remove the books from the top one by one to reach the second last book. Similarly to add a book in a pile it should be added on the top . This principle is the book added in the last should be removed first from the pile. that is "Last In First Out". Stack follows this principle.

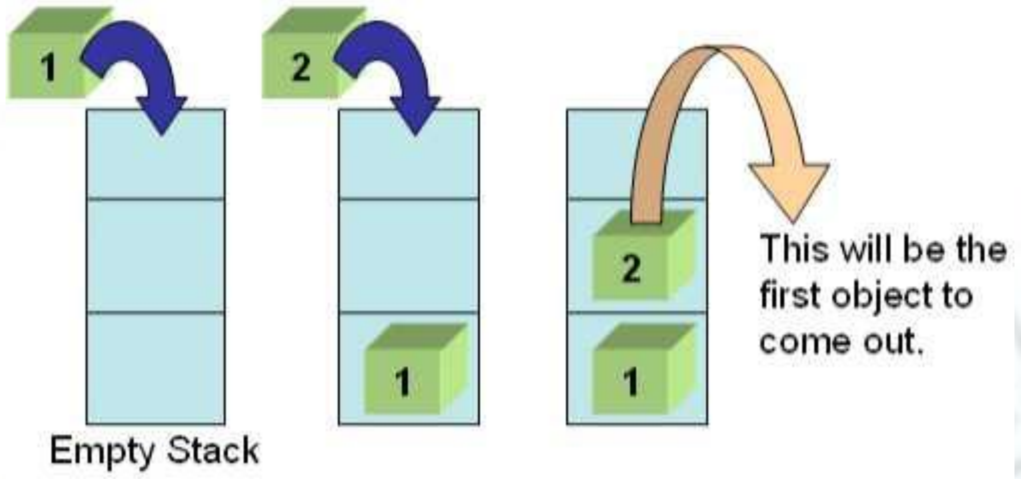
Fig 2.1 Pile of books

There are many other real life stacks such as pile of plates, stack of files, stack of boxes, stack of pennies and stack of trays etc.



Source: wisetome.com/splat/category/computer

Fig 2.2 Building a stack

Value addition: Pictorial representation
Stack principle :Last in First out
 1 2 2 1 1 This will be the first object to come out. Empty Stack

Source: http://www.codeproject.com/KB/dotnet/stacks.aspx?msg=1834122

Value addition: Do you know
Who proposes this concept of stack data structures and When?
The stack was first proposed in 1955, and then patented in 1957, by the German Friedrich L. Bauer. The same concept was developed independently, at around the same time, by the Australian Charles Leonard Hamblin
Source: Wikipedia

2.2 Operations on Stack

Stack can contain different type of elements. We can use stack of integers, stack of characters etc. Number of operations can be done on the stack of any type. The two main operations are *push* and *pop.*

Operations	Description
push	Adding an element
Pop	Deleting an element
Isempty	Check for empty stack
top	Finding top element
Create	Creating a stack

Table 2.1 Stack and Operations

Push Operation adds the element on the top of the stack and also increment the value of the pointer *top.* It hides other elements below the *top* in the stack . If stack is empty, by pushing an element it initialize stack.

Pop Operation removes an element from the top of the stack. After removal it decrements the pointer *top* . The stack pointer only refers to the elements present in the stack. So, removal will not physically remove the element but pointer is moved one down to signify new top of stack after removal.

Top Operation returns the top element of the stack at that time. It will not remove the element but just read it and return.

IsEmpty Operation will check the status of stack at that time i.e whether it is having some elements or not. It will print the message accordingly and return. Similarly we can write the ***IsFull*** operation if needed.

Create Operation will build new stack with no elements whose top pointer is pointing to initial value (-1). Now, it is ready to accept the data .

IsFull Operation will check the status of stack at that time that is whether it is having a space to enter new element or not. It will print the message accordingly and return.

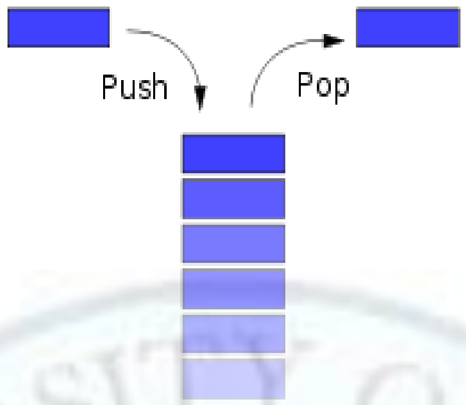
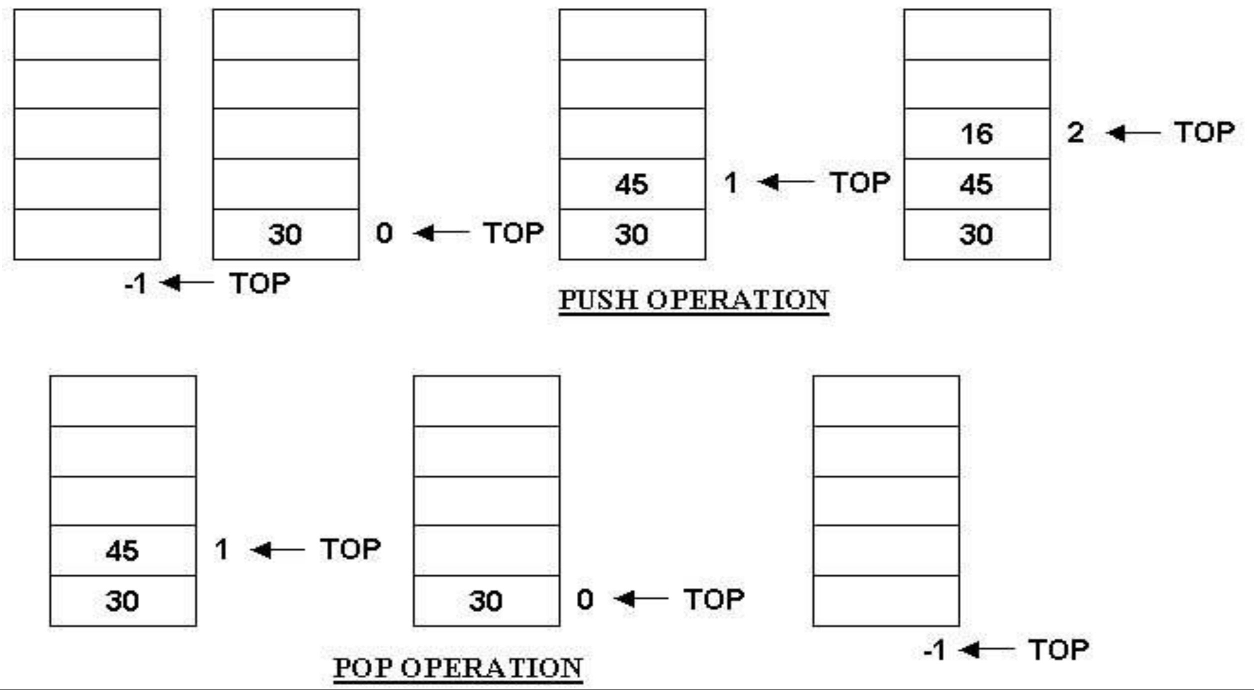


Fig 2.3 Simple stack representation Source: Wikipedia

Value addition: Pictorial representation
Stack operations: push and pop


Source: www.**c4swimmers.esmartguy.com/aboutstack.htm**

Example : The following table shows a series of stack operations and its effect on initially created empty stack of integers.

Operation	Output	Stack Contents
Push(9)	-	(9)
Push(3)	-	(9,3)
Pop()	3	(9)
Push(7)	-	(9,7)
Pop()	3	(9)
Pop()	9	()
isEmpty()	true	()

Source: Self

Stack Algorithms: push and pop

ALGORITHM : STACK

Procedure PUSH(S,TOP,X) : This procedure inserts an element X to the top of the stack which is represented by a vector S consisting MAX elements with a pointer TOP denoting the top element in the stack.

```
STEP 1 : [Check for stack underflow]
If (TOP>=max-1)
then write(stack overflow)
Return
STEP 2 : [Increment TOP]
TOP <-- TOP+1
STEP 3 : [Insert element]
S[TOP] <-- X
STEP 4 : [Finished]
Return
```

The first step of this algorithm checks for an overflow condition. If such a condition exists, then the insertion can't be performed and an appropriate error message results.

Procedure POP(S,TOP,X) : This procedure removes top element from the stack.

```
STEP 1 : [Check for the underflow on stack]
If (TOP<0)
then write(stack underflow)
```

```
Return
STEP 2 : [Hold the former top element of stack into X]
X <-- S[TOP]
STEP 3 : [Decrement the TOP pointer or index by 1]
TOP <-- TOP-1
STEP 4 : [finished-Return the popped item from the stack]
Return(X)
```

As Underflow condition is checked for in the first step of the algorithm. If such a condition exists, then the deletion cannot be performed and an appropriate error message results.

Procedure Display(S,TOP) : This procedure displays the contents of the stack i.e., vector S.

```
STEP 1 : [check for empty on stack]
if (TOP==-1)
then write('stack empty')
Return
STEP 2 : [Repeat through STEP 3 from i=TOP to 0]
Repeat through STEP 3 for i=TOP to 0 STEP-1]
STEP 3 : [Display the stack content]
write (S[i])
STEP 4 : [Finished]
Return
```

The first step of this algorithm checks for an empty condition. If such a condition exists, then the contents of the stack cannot be displayed and an appropriate error message results.

Source: www.**c4swimmers.esmartguy.com/aboutstack.htm**

2.3 Software Stacks

Stack can be used in high level language. In the program, we can use only two operations push and pop. They are implemented in high level language with two representations linear and linked.

2.3.1 Implementation of Stack

Stack can be implemented in two representations Linear and Linked which is taken as a interface and only user is allowed to do operations push and pop. We can implement stack in high level language like C,C++.

2.3.1.1 Linear Representation

Linear representation of stack can be using an array. To implement a Linear stack of size 10 elements, we need a variable, called top, that holds the index of the top element of the stack and an array ,named elements to hold the maximum 10 elements of the stack.

In programming language C++, Linear stack is represented in the form of class structure as
below:

```
class stack
{
 int elements[10];
 int top;
 public:
 stack() { top=-1;}
 void push (int ele);
 int pop();
};
```

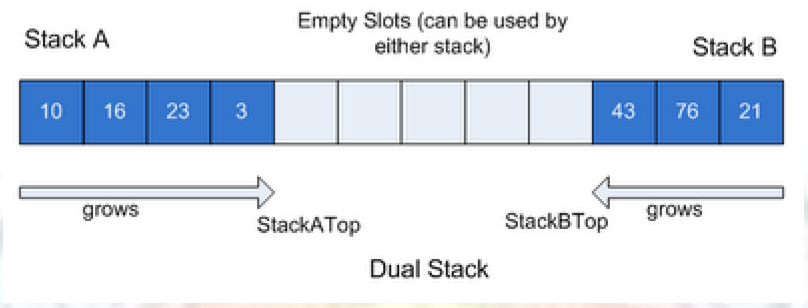
The functions for pushing and popping elements are given below:

```
//pushing elements into stack at top
 void stack::push(int ele)
{ if(top==10)
  {
  cout<<"Overflow...";
  }
  else
  { top++;

  elements[top]=ele;
  }
}
// popping elements from top
int stack::pop()
{ if(top==-1)
  {
  cout<<"UNDERFLOW...";
  return -1;
  }
  else
  {  int m=elements[top];
     top--;
     return m;
  }}
```

Value addition: Frequently asked question

How to represent two stacks using one array

Visual Representation of the two stacks in the array.



The obvious solution is to have the two stacks at the two ends of the array. The stacks will grow in opposite direction. When the two stacks collide and there is no more room in the array the stacks will over flow

Source: www.technicalinterviewquestions.net/2009/04/implement-two-2-stacks-one-1-array.html

2.3.1.2 Linked Representation

The array based representation of stacks has following limitations :

1. size of the stack must be known in advance.
2. We may come across situations when an attempt to push an element cause overflow. However, stack, as an abstract data structure cannot be full .Hence abstractly, it is always possible to push an element onto stack. Therefore, representing stack as an array prohibits the growth of stack beyond the finite number of elements.

Therefore , some dynamic structure should be used to represent stack.

A stack represented using a linked list is also known as linked stack. Linked stack can be represented as singly linked list or doubly linked list. The linked representation allow a stack to grow to a limit of the computer’s memory. The class structures for linked stack is given as follows..

```
class node
{ public:
  int info;
  node *next;

  node()
  { next=0;
  }
  node(int x,node *y=0)
  { info=x;
    next=y;

  }
};

class stack
{ node *top;
  public:
  stack(){top=0;}
  void push(int ele);
  void traverse();
  int pop();
};
```

Node of linked list is the class having integer element and next is a pointer pointing to next element. Class stack contains top pointer pointing to first element of linked list and all related operations as member functions . The push and pop operations are exactly same as discussed earlier except list is maintained by next pointer. Traverse operation prints all the elements from the stack.The *pop* operation deletes the first node from the linked list by performing the following task:

- Store the *top* in some temporary pointer
- Store the information of top element in some variable
- If stack contains only one element, top should initialize to null.
- Otherwise, change the *top* pointer to the second element of linked list
- Free the node pointed by temporary pointer

```
int stack::pop()
{  node *t;
   int m=top->info;
   t=top;
   if ( top->next==0) { top=0; }
  else
  { top=top->next;}
   delete t;
   return m;
}
```

The *push* operation adds the new node in the beginning of linked list by performing the following tasks:

- Create the new node using the variable passed as an argument
- Check the stack is empty or not
- If stack is empty, the new node pointer becomes the *top* of the list.
- If stack is not empty, the next pointer of new node should points to the current *top* and then new node becomes the new *top* pointer.

```
void stack::push(int ele)
{ node *p;
 p=new node(ele);
  if(top==0)
  { top=p;
  }
  else
  {
   p->next=top;
    top=p;
    }
}
```

The *traverse* operation will print all the information from the nodes of the list by performing the following task:

- Start the traverse by node pointed by *top*
- Use the loop to print the information of each node
- After printing the information of the current node , advance the pointer to the next node using the *next* field of the node

```
void stack::traverse()
{ node *q;
  q=top;
  cout<<"Entered info is.......";
  while(q!=0)
  { cout<<q->info<<" ";
    q=q->next;
  }
}
```

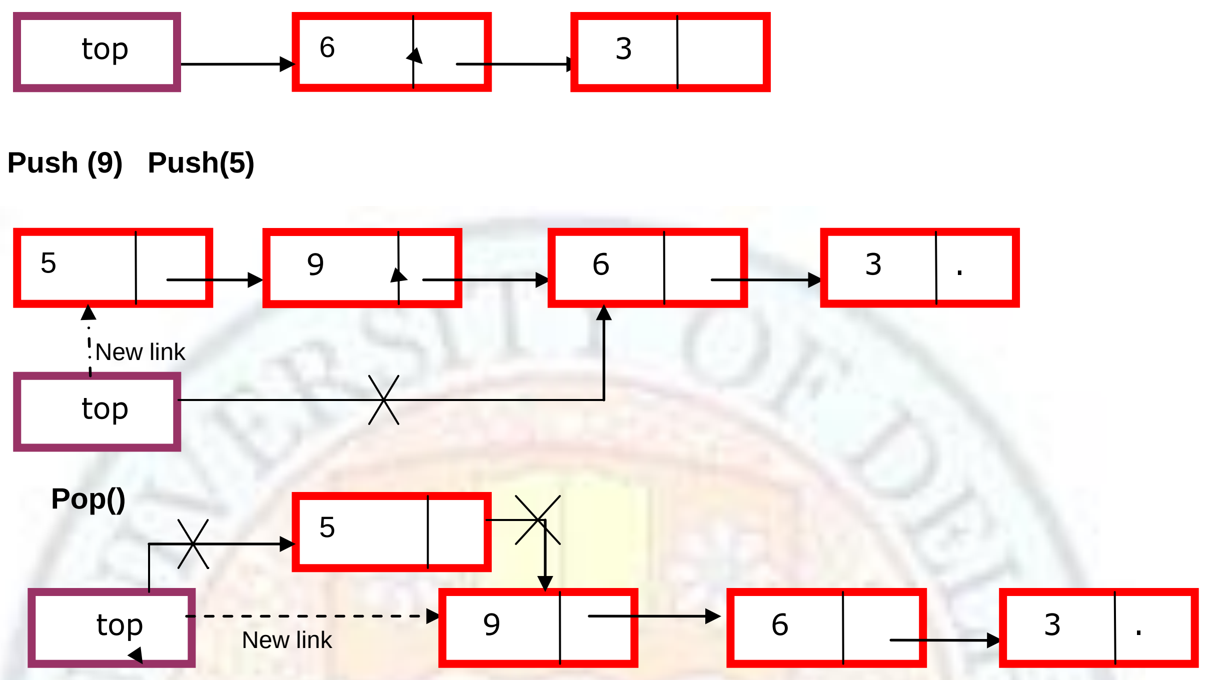


Fig 2.4 : Linked representation of stack : push and pop
Source:self

2.3.1.3 Comparing Linear and Linked Representations of Stack

When we compare two implementations of stack array and Linked, we consider certain factors. So which implementation is better? The answer, as usual, is: It depends. The linked implementation certainly gives more flexibility, and in applications where the number of stack items can vary greatly, it wastes less space when the stack is small. In situations where the stack size is totally unpredictable, the linked implementation is preferable, because size is largely irrelevant. Why, then, would we ever want to use an array-based implementation? Because it is short, simple, and efficient. If pushing and popping occur frequently, the array-based implementation executes more quickly because it does not incur the run-time overhead of the new and delete operations.

Overall, the three stack implementations(Static Array, Dynamic array and Linked representation) are roughly equivalent in terms of the amount of work they do, differing in only one of the five operations and the class destructor(which is used in linked representation for destroying the pointers and nodes after use). Note that if the difference had occurred in the Push, Top, or Pop operation, rather than the less frequently called destructor, it would be more significant. Table 5.2 summarizes the Big-O comparison of the stack operations.

Table 2.2 : Big-O Comparison of Stack Operations

	Static Array Implementation	**Dynamic Array Implementation**	**Linked Implementation**
Class constructor	O(1)	O(1)	O(1)
IsFull	O(1)	O(1)	O(1)
IsEmpty	O(1)	O(1)	O(1)
Push	O(1)	O(1)	O(1)
Pop	O(1)	O(1)	O(1)
Top	O(1)	O(1)	O(1)
Destructor	**NA**	**O(1)**	**O(N)**

2.4 Generic Stack

A Generic stack is a C++ language construct that allows the compiler to generate multiple versions of a class type or a function by allowing parameterized types. This can be achieved using *template parameters*.

```
template<class stacktype>
class stack
{
 stacktype elements[10];
 int top;
 public:
 stack() { top=-1;}
 void push (stacktype ele);
 stacktype pop();
};
```

The main program code creates three different types of stack integer, float, character using the following statements:

```
stack<int> s1;
stack<float> s2;
stack<char> s3;
```

The generic functions can be written as follows:

```
template<class stacktype>
void stack<stacktype>:: push (stacktype ele);
{…………..}

template<class stacktype>
stacktype stack<stacktype>:: pop (void);
{…………..}
```

Value addition: Do You Know

How is stack represented in other languages?

Some languages, like LISP and Python, do not call for stack implementations, since **push** and **pop** functions are available for any list. All Forth-like languages (such as Adobe PostScript) are also designed around language-defined stacks that are directly visible to and manipulated by the programmer.

C++'s Standard Template Library provides a "stack" templated class which is restricted to only push/pop operations. Java's library contains a Stack class that is a specialization of Vector---this could be considered a design flaw, since the inherited get() method from Vector ignores the LIFO constraint of the Stack.

Source: Wikipedia

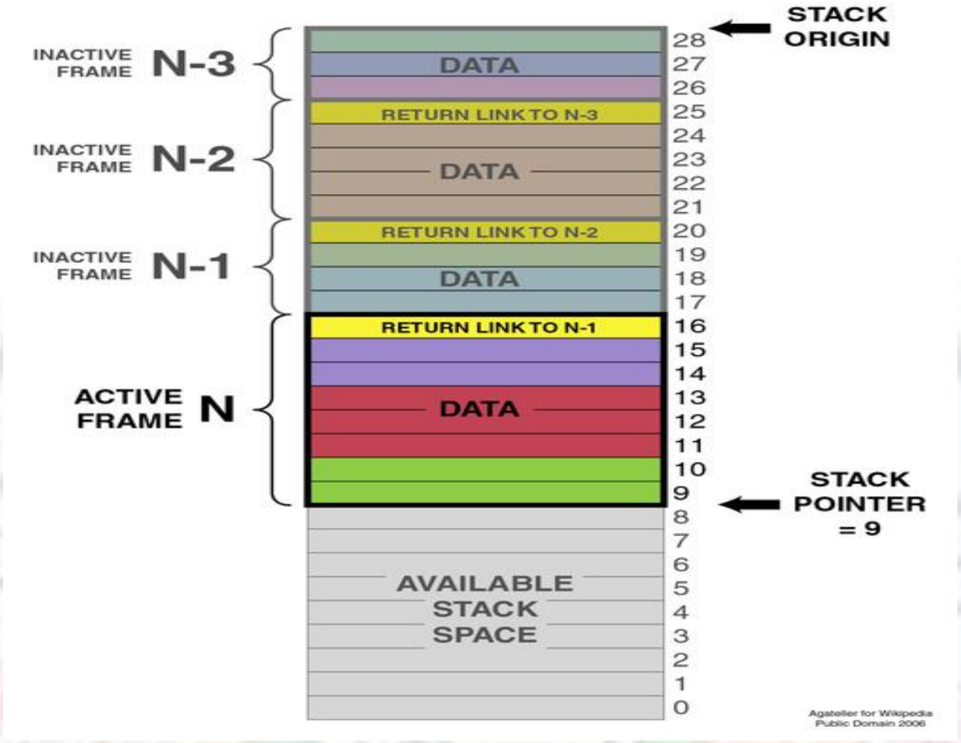
2.5 Hardware Stacks

2.5.1 Architecture of Stack

A typical stack is an area of computer memory with a fixed origin and a variable size. Initially the size of the stack is zero. A *stack pointer,* usually in the form of a hardware register, points to the most recently referenced location on the stack; when the stack has a size of zero, the stack pointer points to the origin of the stack. The two operations applicable to all stacks are: *push* and *pop.*

There are many variations on the basic principle of stack operations. Every stack has a fixed location in memory at which it begins. As data items are added to the stack, the stack pointer is displaced to indicate the current extent of the stack, which expands away from the origin (either up or down, depending on the specific implementation).For example, a stack might start at a memory location of one thousand, and expand towards lower addresses, in that case new data items are stored at locations ranging below 1000, and the stack pointer is decremented each time a new item is added. When an item is removed from the stack, the stack pointer is incremented. Hardware support can be given as stack in main memory, registers or dedicated memory.

Fig 2.5 Hardware stack



source:http://www.absoluteastronomy.com/topics/Stack_(data_structure)

3 Stacks: Applications

Stack is such a data structure which is used as software stack and hardware stack in various concept related with computers.

3.1 Delimiter Matching

One of the main applications of stack in compilers deal with determining whether delimiters in an input string or files are matched or not. As an example, consider the language of parentheses. A sequence of symbols involving parenthesis(), addition(+), multiplication(*) and integers, is said to have balanced-parentheses if each right parenthesis has a corresponding left parenthesis that occurs before it. For example: These expressions have balanced parentheses:

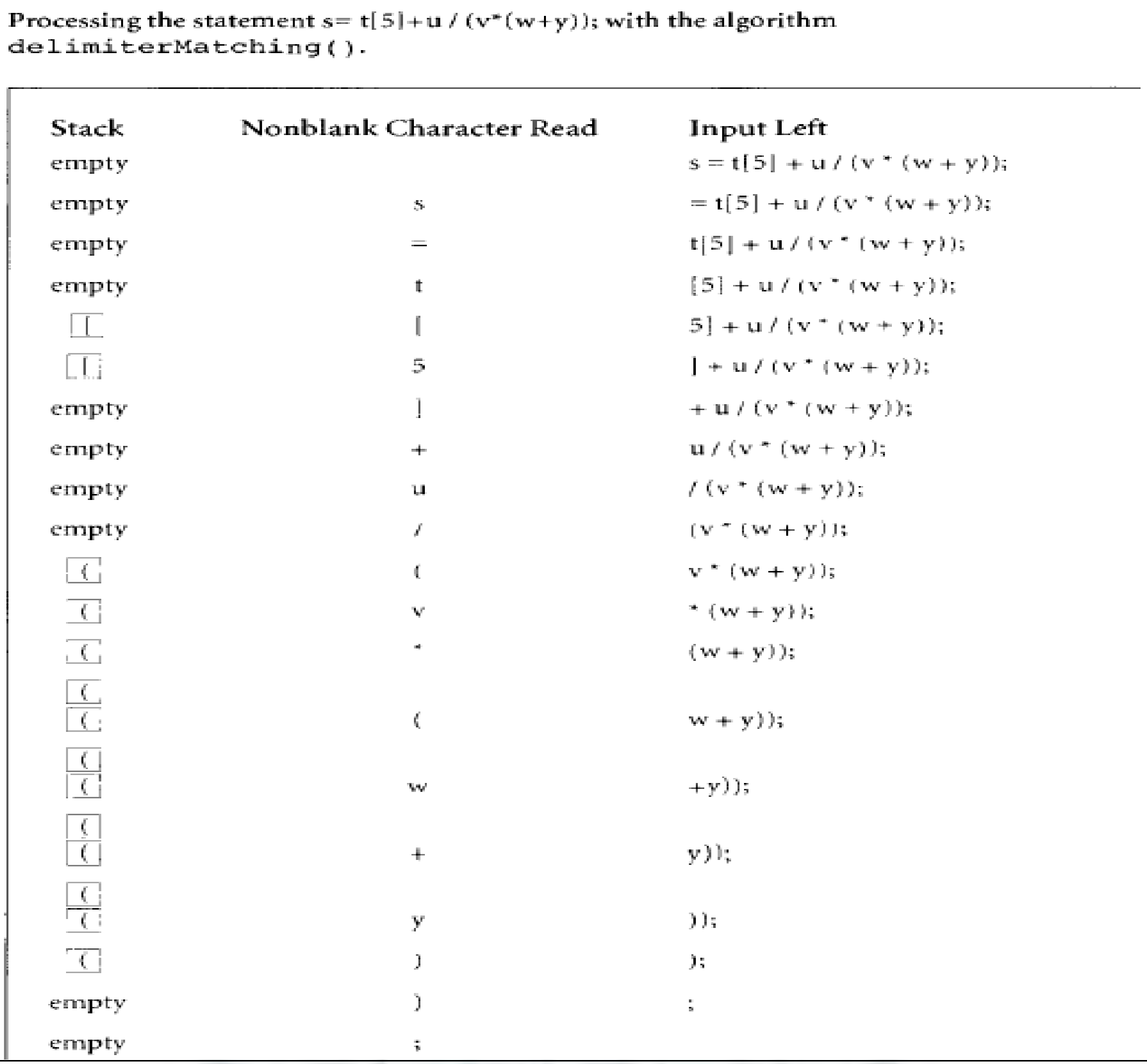
```
2*7          // no parenthesis - still balanced
(1+3)
((2*16)+1)*(44+(17+9))
```

These following expressions do not have balanced parenthesis:
(44+38
)   // a right parenthesis with no left parenthesis

```
(55+(12*11)  // missing a )
```

Value addition: Illustration

How is stack used for Delimiter matching?



Processing the statement s= t[5]+u / (v*(w+y)); with the algorithm `delimiterMatching()`.

Stack	Nonblank Character Read	Input Left
empty		s = t[5] + u / (v * (w + y));
empty	s	= t[5] + u / (v * (w + y));
empty	=	t[5] + u / (v * (w + y));
empty	t	[5] + u / (v * (w + y));
[	[	5] + u / (v * (w + y));
[	5	] + u / (v * (w + y));
empty	]	+ u / (v * (w + y));
empty	+	u / (v * (w + y));
empty	u	/ (v * (w + y));
empty	/	(v * (w + y));
(	(	v * (w + y));
(	v	* (w + y));
(	*	(w + y));
((	(	w + y));
((	w	+y));
((	+	y));
((	y	));
(	)	);
empty	)	;
empty	;	

Source: Data structures and algorithms in C++ by Adam Drozdek

We use stack to do this matching for storing the characters.

- Start with an empty stack.
- For each left parenthesis, push left parenthesis.
- For each right parenthesis, pop left parenthesis.
- For each non-parenthesis character, do nothing.
- The expression is balanced if the stack is empty and there are no more characters to process.

- It is not balanced if either after the last character the stack is not empty (too many left parenthesis), or if the stack is empty and a right paren is encountered (a right without a left).

. We can match multiple types of delimiters. For example, we might want to match (), [], and {}. In that case we can push the left delimiter onto the stack, and, when we pop, do a check, as in the pseudocode.

```
if (next character is a right delimiter) then {
  if (stack is empty) then
    return false
  else {
    pop the stack
    if (right delimiter is corresponding version
        of what was popped off the stack) then
             continue processing
    else
       return false
    }
}
```

3.2 Expression Solving

The computer usually evaluates an arithmetic expression written in infix notation (A+B*C) in two steps. First it converts the expression to postfix expression (ABC*+), and then evaluates the postfix expression. In each step the stack is the main tool to accomplish the task.

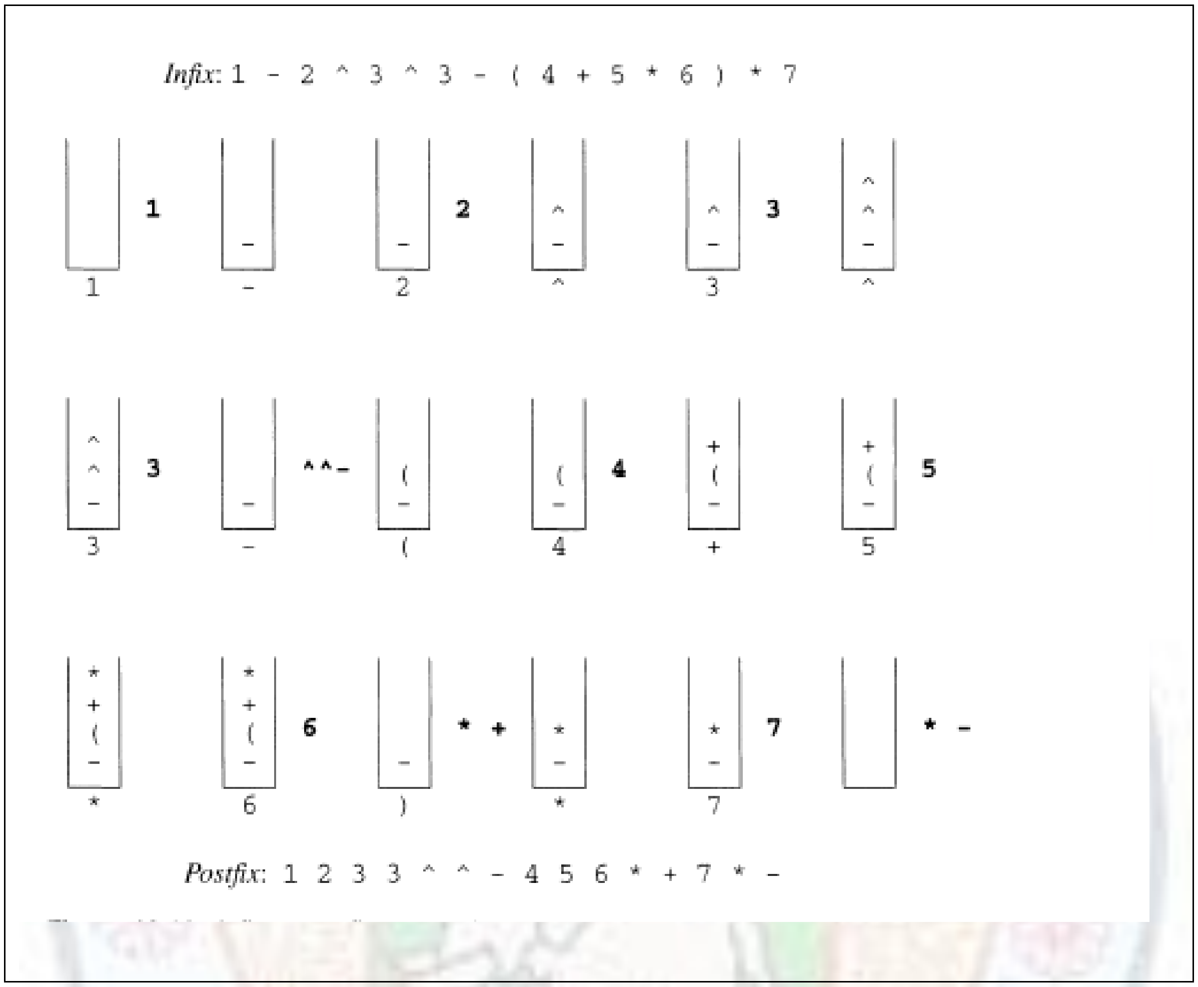
For Example to evaluate expression (1-2^3^3- (4+5*6) * 7):

- First convert infix expression into postfix expression as follows

  1 2 3 3 ^ ^ - 4 5 6 * + 7 * -

- Evaluate this postfix expression using stack
  Result: - 8
  These steps will be explained in next section.

Value addition: Illustration
How is stack used for conversion of infix to postfix?


Infix: 1 - 2 ^ 3 ^ 3 - (4 + 5 * 6) * 7
Postfix: 1 2 3 3 ^ ^ - 4 5 6 * + 7 * -

Source: Data structure and problem solving using C++ by Mark Weiss

<u>**Algorithm for converting Infix to postfix expression:-**</u>

Step-1: Check whether the current element in the expression is an operator or operand. If its an operand then go to step-2 or else step-3

Step-2: Put the element in the Postfix output stream and go for the next element in the expression, if any

Step-3: a. find the priority of that operator
b. if the operator's priority > priority of Stack's top then push that operator inside the stack and go for the next element; or else pop the stack, write that value to the Postfix output stream and go to start of this step

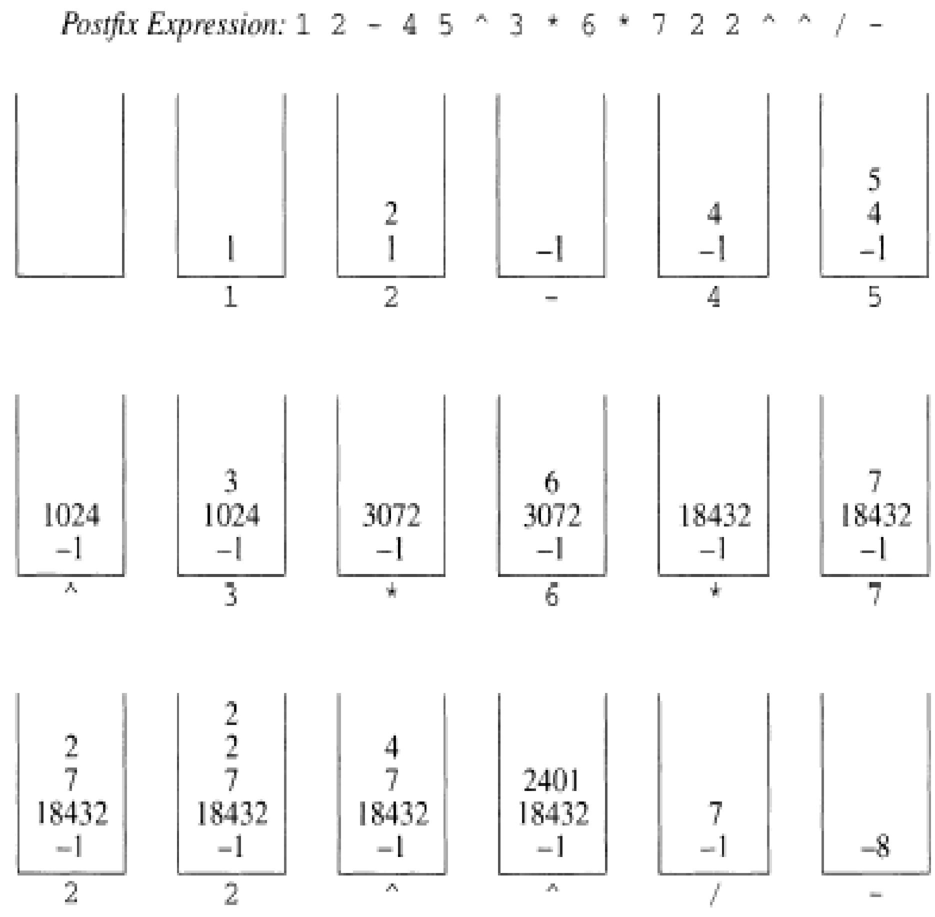
Step-4: Empty the Stack and put the popped values to the output stream directly

Value addition: Source Code

How is stack used in conversion of infix to prefix

```
/* PROGRAM TO CONVERT INFIX TO PREFIX EXPRESSION USING STACK */
#include<stdio.h>
#include<string.h>

/*MACRO FUNCTION TO CHECK WHETHER GIVEN CHARACTER IS OPERAND OR
NOT */
#define operand(x)(x>='a'&&x<='z' || x>='A'&&x<='Z' ||
x>='0'&&x<='9')
char infix[30],prefix[30],stack[30];
int top,i=0;

/* FUNCTION TO INITIALIZE THE STACK */
void init()
{
        top=-1;
}

/* FUNCTION TO PUSH AN OPERATOR ON TO THE STACK */
void push(char x)
{
        stack[++top]=x;
}

/* FUNCTION TO POP A CHARACTER STORED ONTO THE STACK */
char pop()
{
        return(stack[top--]);
}

/* FUNCTION TO RETURN IN STACK PRIORITY OF A CHARACTER */
int isp(char x)
{
int y;
y=(x=='('?6:x=='^'?4:x=='*'?2:x=='/'?2:x=='+'?1:x=='-
'?1:x==')'?0:-1);
return y;
}

/* FUNCTION TO RETURN INCOMING CHARACTER'S PRIORITY */
int icp(char x)
{
int y;
y=(x=='('?6:x=='^'?4:x=='*'?2:x=='/'?2:x=='+'?1:x=='-
'?1:x==')'?4:-1);
return y;
}
```

```
/* FUNCTION TO CONVERT THE GIVEN INFIX TO PREFIX EXPRESSION */
void infixtoprefix()
{
        int j,l=0;
        char x,y;
        stack[++top]='\0';
        for (j=strlen(infix)-1; j>=0; j--)
                {
                        x=infix[j];
                        if (operand(x))
                                prefix[l++]=x;
                        else
                                if (x=='(')
                                        while ((y=pop())!=')')
                                                prefix[l++]=y;
                                else
                                        {
                                                while
(isp(stack[top])>=icp(x))

        prefix[l++]=pop();
                                                push(x);
                                        }
                }
        while (top>=0)
                prefix[l++]=pop();
}

/* MAIN PROGRAM */
int main()
{
  init();
  printf("Enter an infix expression :\n");
  scanf("%s",infix);
  infixtoprefix();
  strrev(prefix);
  printf("The resulting prefix expression is %s",prefix);
  return 0;
} // End of main
```

Value addition: Illustration
How is stack used for evaluation of postfix expression?
 Source: Data structure and problem solving using C++ by Mark Weiss

Algorithm for evaluating postfix expression

STEP 1 : Read the given postfix expression into a string called postfix.

STEP 2 : Read one character at a time & perform the following operations :
1. If the read character is an operand, then convert it to float and push it onto the stack.
2. If the character is not an operand, then pop the operator from stack and assign to OP2. Similarly, pop one more operator from stack and assign to OP1.
3. Evaluate the result based on operator x.
4. Push the result (based on operator x) onto the stack.

STEP 3 : Repeat **STEP 2** till all characters are processed from the input string.

STEP 4 : Return the result from this procedure or algorithm and display the result in main program.

3.3 Adding Two Large Numbers

Stack can be used to add two large numbers. As the memory limits the storage of large numbers also large number operations require double the space for storage. It is needed a data structure which store the large numbers digit by digit. Stack is data structure which stores large numbers digit by digit in specific order. Then the LIFO principle helps to add unit place digit and store the result in third stack. This process repeats till the stacks are empty.

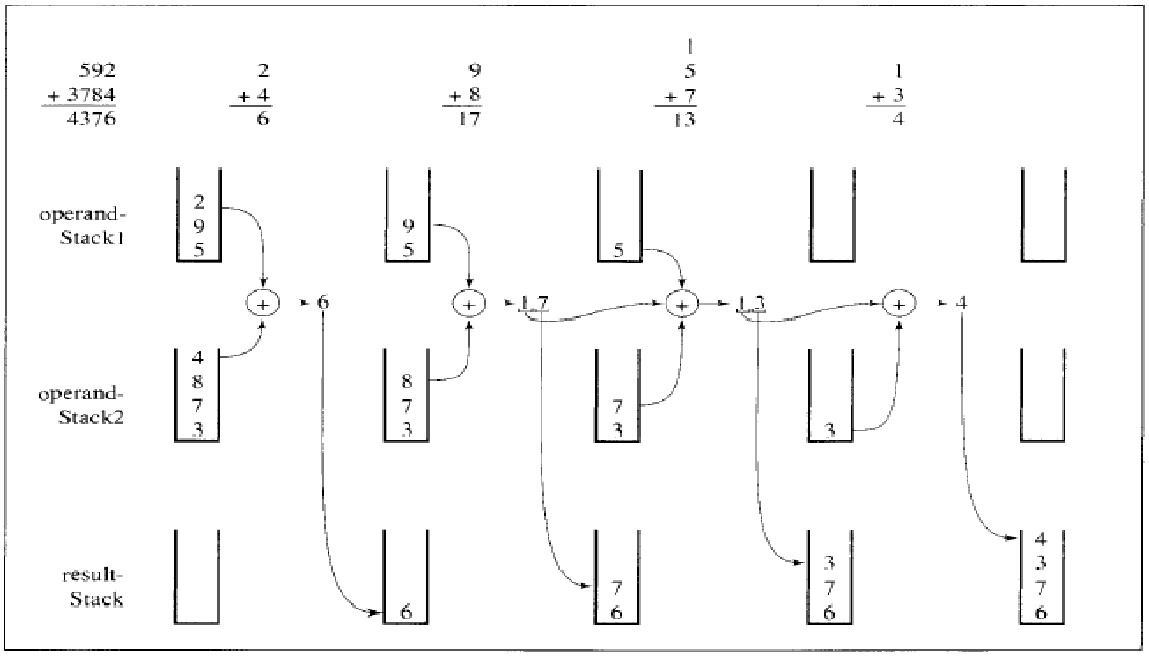
For example : To add 592 and 3784 ,we follow the following steps:

- Take two stacks for storing two numbers stack1 and stack2.
- Numbers can be stored in the stack by putting digit with highest place value first so that digit at ones place should be on the top of stack.
- Now the two stacks contains digits in the order depicted in the diagram.
- Pop the two stacks and add the digits from stacks with the carry which is initially zero. Push the result in the third stack digit by digit.
- Repeat these steps of popping and adding digits with carry until the two stacks were empty.

Value addition: Pictorial Representation

How to add two numbers 592 and 3784 using stack?

Source: Data structures and algorithms in C++ by Adam Drozdek



Value addition: Algorithm

How to add two large numbers?

```
addingLargeNumbers()
    read the numerals of the first number and store the numbers corresponding to
    them on one stack;
    read the numerals of the second number and store the numbers corresponding
    to them on another stack;
    result = 0;
    while at least one stack is not empty
        pop a number from each nonempty stack and add them to result;
        push the unit part on the result stack;
        store carry in result;
    push carry on the result stack if it is not zero;
    pop numbers from the result stack and display them;
```

Source: Data structures and algorithms in C++ by Adam Drozdek

3.4 Stack in Recursion

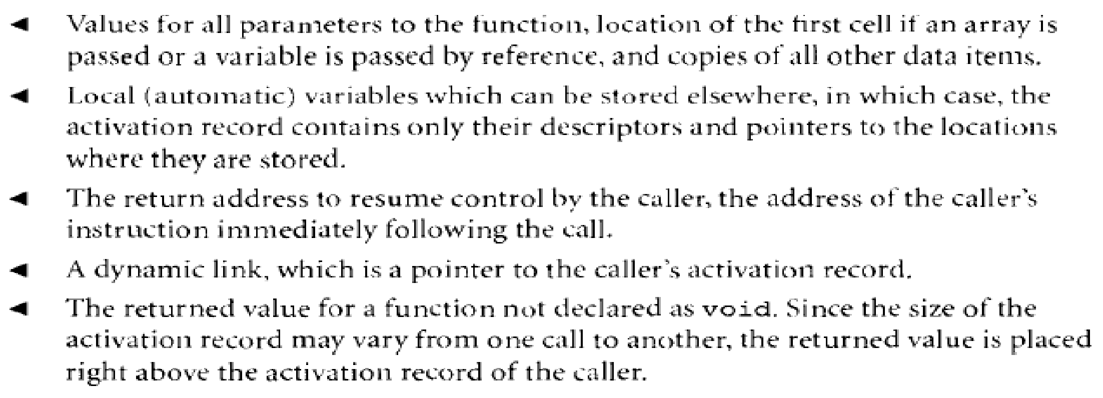
As we have studied earlier about *recursion* it is briefly described as - when a function calls itself - represents yet another kind of looping. Recursive operations are neither indexed nor easily converted to something that is (such was the case with for..in and for..each loops). This requires a solution separate to those used before.

```
// pseudo code
function recursive(){
        recursive();
}
recursive();
```

When functions call one another (including themselves), they are added to the *call stack*. The call stack contains all of the functions currently being called. When a function completes its execution or returns a value, it is removed from the call stack and the function which called it resumes executing. Recursive functions add themselves to the call stack repeatedly until some condition is reached that the last function returns a value rather than calling itself again.

Value addition: Do you know

What are the contents of activation record stored in stack?



◄ Values for all parameters to the function, location of the first cell if an array is passed or a variable is passed by reference, and copies of all other data items.

◄ Local (automatic) variables which can be stored elsewhere, in which case, the activation record contains only their descriptors and pointers to the locations where they are stored.

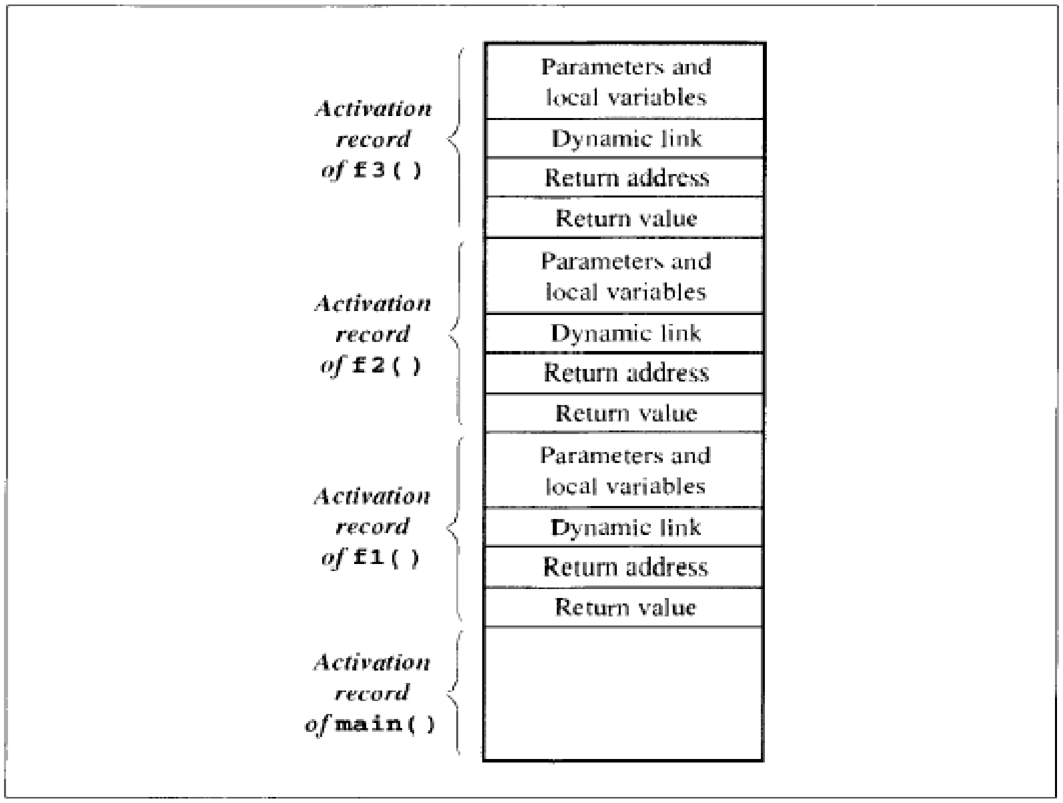
◄ The return address to resume control by the caller, the address of the caller's instruction immediately following the call.

◄ A dynamic link, which is a pointer to the caller's activation record.

◄ The returned value for a function not declared as `void`. Since the size of the activation record may vary from one call to another, the returned value is placed right above the activation record of the caller.

Source: Data structures and algorithms in C++ by Adam Drozdek

Contents of the run-time stack when main() calls function f1(), f1() calls f2(), and f2() calls f3().

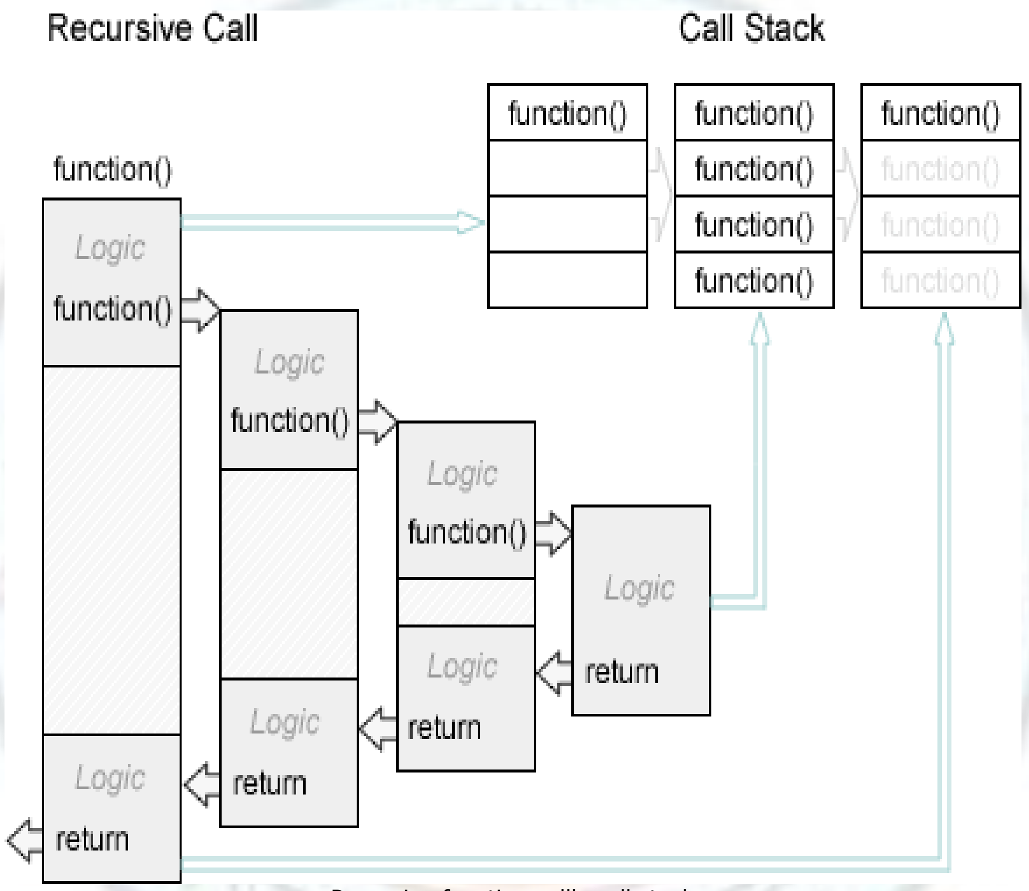

Activation record of f3()
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of f2()
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of f1()
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of main()

Source: Data structures and algorithms in C++ by Adam Drozdek
Fig 2.6: Stack contents showing function calls in the order main()->f1()->f2()->f3()

This is how method calls (recursive and non-recursive) really work:

- At runtime, a stack of activation records (ARs) is maintained: one AR for each active method, where "active" means: has been called, has not yet returned.
- Each AR includes space for:
  - the method's parameters,
  - the method's local variables,
  - the return address -- where (in the code) to start executing after the method returns.

- When a method is called, its AR is pushed onto the stack. The return address in that AR is the place in the code just after the call (so the return address is the "marker" for the previous "clone").
- When a method is about to return, the return address in its AR is saved, its AR is popped from the stack, and control is transferred to the place in the code referred to by the return address.



Recursive function call's call stack

Source: www.senocular.com/.../asyncoperations/?page=2

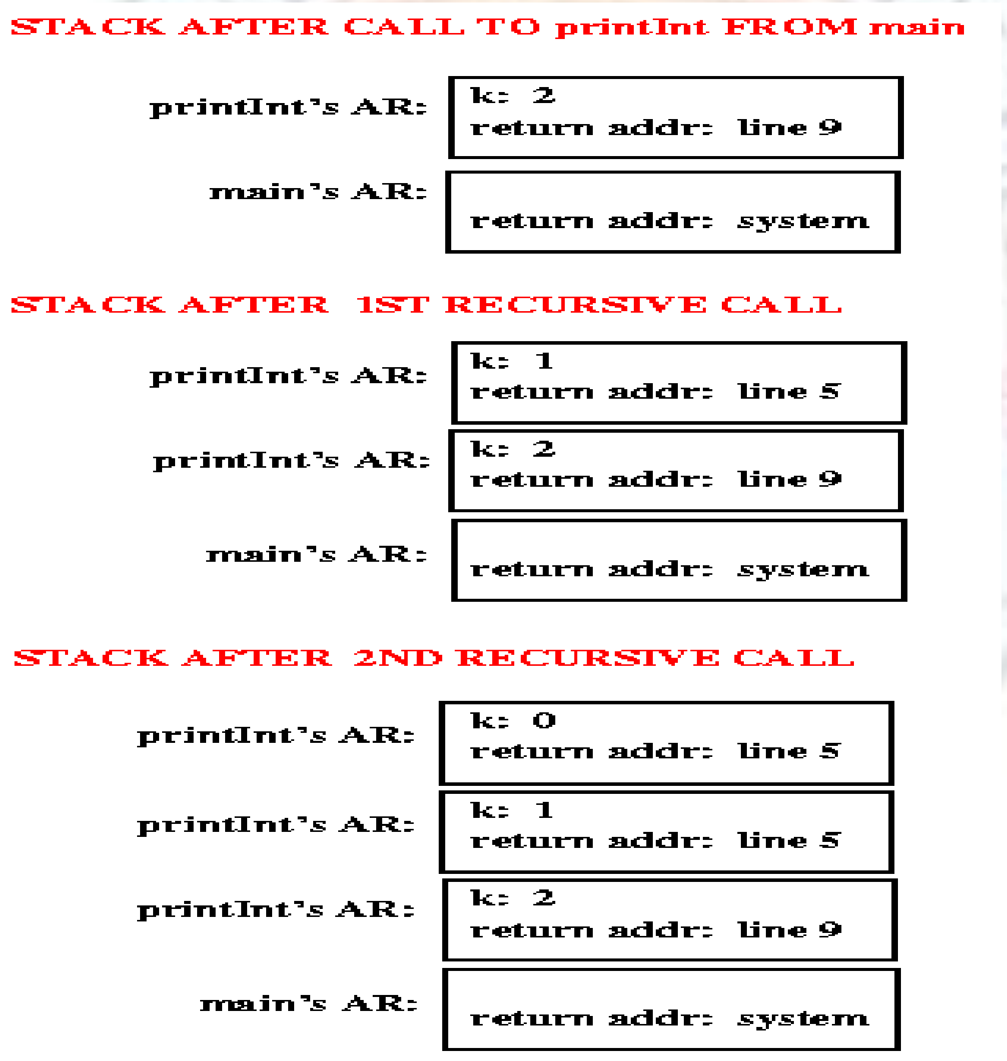
Fig 2.7 : recursive function call stack

Value addition: ILLustration

How is stack used in implementation of recursion?

```
1. void printInt( int k ) {
2.     if (k <= 0) return;
3.     System.out.println( k );
4.     printInt( k - 1 );
5. }
6.
7. void main(...) {
8.     printInt(2);
9.}
```

The following pictures illustrate the runtime stack as this program executes.



Note that all of the recursive calls have the same return address -- after a recursive call returns, the previous call to *printInt* starts executing again at line 5. In this case, there is nothing more to do at that point, so the method would, in turn, return.

Note also that each call to *printInt* causes the current value of k to be printed, so the output of the program is: 2 1.

Now consider a slightly different version of the *printInt* method:

```
1.  void printInt (int k) {
2.      if (k<=0) return;
3.      printInt (k-1);
4.      System.out.println(k)
5.  }
```

Now what is printed as a result of the call *printInt(2)*? Because the print statement comes **after** the recursive call, it is not executed until the recursive call finishes (i.e., *printInt*'s activation record will have line 4 -- the print statement -- as its return address, so that line will be executed only after the recursive call finishes). In this case, that means that the output is: 1 2 (instead of 2 1, as it was when the print statement came before the recursive call)..

Source: http://pages.cs.wisc.edu/~vernon/cs367/notes/6.RECURSION.html

3.5 Reverse a string

Stack can be used for reversing a string . Stack is a data structure which works on LIFO principle . This property of stack is utilized while reversing a string. We can input the string and extract the individual characters from it by using some function. The characters can be stored in the stack as extracted from the string i.e leftmost character is first character to go into the stack. All the characters are added so that the rightmost character should be on the top. According to LIFO principle, Pop function will remove the rightmost character first from the top which can be added to the string to get reverse string .

For Example : "abcdefghijk"

Will be entered in the stack with Push(a), Push(b),…………so on

as a b c d e f g h i j k --→ Top

pop()-→ will return ' k', Pop()---→ 'j', ……………. So on

Value addition: Source Code
Reversing a string using stack

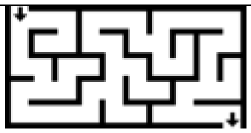
```
// reverse.java
// stack used to reverse a string
// to run this program: C>java ReverseApp
import java.io.*;                  // for I/O
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;
   private char[] stackArray;
   private int top;

//--------------------------------------------------------------
-
   public StackX(int max)     // constructor
      {
      maxSize = max;
      stackArray = new char[maxSize];
      top = -1;
      }

//--------------------------------------------------------------
-
   public void push(char j)  // put item on top of stack
      {
      stackArray[++top] = j;
      }

//--------------------------------------------------------------
-
   public char pop()         // take item from top of stack
      {
      return stackArray[top--];
      }

//--------------------------------------------------------------
-
   public char peek()        // peek at top of stack
```

```
      {
      return stackArray[top];
      }

//--------------------------------------------------------------
-
   public boolean isEmpty()   // true if stack is empty
      {
      return (top == -1);
      }

//--------------------------------------------------------------
-
   }  // end class StackX

////////////////////////////////////////////////////////////////

class Reverser
   {
   private String input;                // input string
   private String output;               // output string

//--------------------------------------------------------------
-
   public Reverser(String in)           // constructor
      { input = in; }

//--------------------------------------------------------------
-
   public String doRev()                // reverse the string
      {
      int stackSize = input.length();   // get max stack size
      StackX theStack = new StackX(stackSize);  // make stack

      for(int j=0; j<input.length(); j++)
         {
         char ch = input.charAt(j);     // get a char from
input
         theStack.push(ch);             // push it
         }
      output = "";
      while( !theStack.isEmpty() )
         {
         char ch = theStack.pop();      // pop a char,
         output = output + ch;          // append to output
         }
      return output;
      }  // end doRev()

//--------------------------------------------------------------
-
   }  // end class Reverser

i ////////////////////////////////////////////////////////////////
```

Source: Data Structure And Algorithms In Java - Mitchel Waite

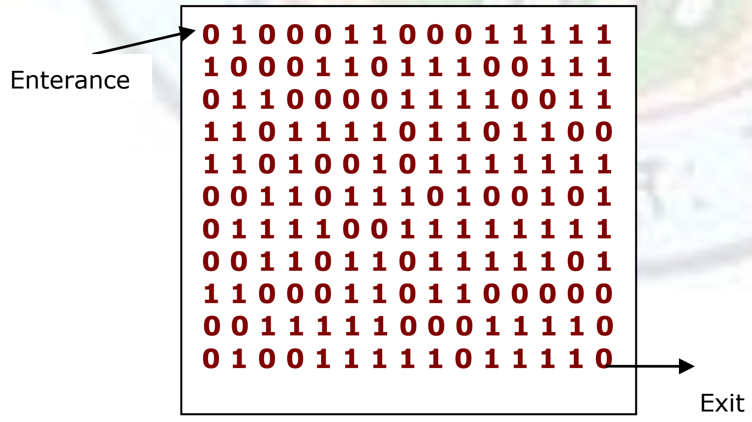
Value addition: Case study

The Maze Problem



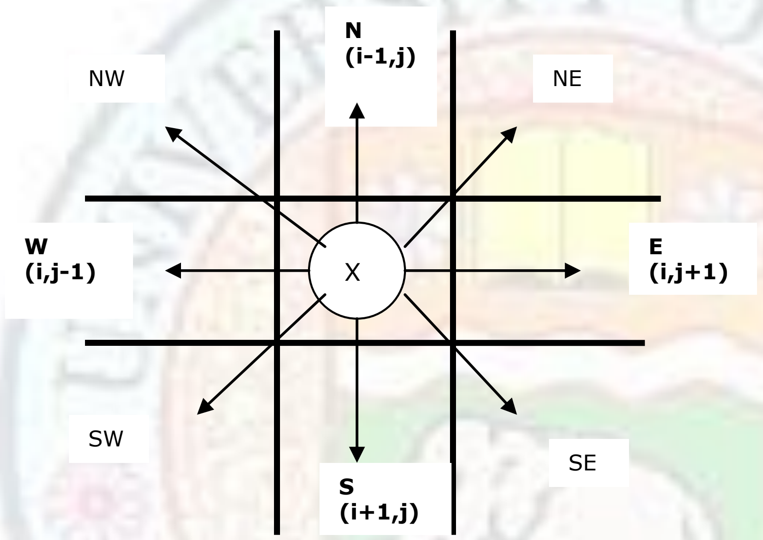
The rat-in-a-maze experiment is a classical one from experimental psychology. A rat (or mouse) is placed through the door of a large box without a top. Walls are set up so that movements in most directions are obstructed. The rat is carefully observed by several scientists as it makes its way through the maze until it eventually reaches the other exit. There is only one way out, but at the end is a nice hunk of cheese. The idea is to run the experiment repeatedly until the rat will zip through the maze without taking a single false path. The trials yield his learning curve.
We can write a computer program for getting through a maze and it will probably not be any smarter than the rat on its first try through. It may take many false paths before finding the right one. But the computer can remember the correct path far better than the rat. On its second try it should be able to go right to the end with no false paths taken, so there is no sense re-running the program. Why don't you sit down and try to write this program yourself before you read on and look at our solution. Keep track of how many times you have to go back and correct something.

Let us represent the maze by a two dimensional array, MAZE(1:m, 1:n), where a value of 1 implies a blocked path, while a 0 means one can walk right on through. We assume that the rat starts at MAZE(1,1) and the exit is at MAZE(m,n).



```
0 1 0 0 0 1 1 0 0 0 1 1 1 1 1
1 0 0 0 1 1 0 1 1 1 0 0 1 1 1
0 1 1 0 0 0 0 1 1 1 1 0 0 1 1
1 1 0 1 1 1 1 0 1 1 0 1 1 0 0
1 1 0 1 0 0 1 0 1 1 1 1 1 1 1
0 0 1 1 0 1 1 1 0 1 0 0 1 0 1
0 1 1 1 1 0 0 1 1 1 1 1 1 1 1
0 0 1 1 0 1 1 0 1 1 1 1 1 0 1
1 1 0 0 0 1 1 0 1 1 0 0 0 0 0
0 0 1 1 1 1 1 0 0 0 1 1 1 1 0
0 1 0 0 1 1 1 1 1 0 1 1 1 1 0
```

With the maze represented as a two dimensional array, the location of the rat in the maze can at any time be described by the row, i, and column, j of its position. Now let us consider the possible moves the rat can make at some point (i,j) in the maze. Figure 3.6 shows the possible moves from any point (i,j). The position (i,j) is marked by an X. If all the surrounding squares have a 0 then the rat can choose any of these eight squares as its next position. We call these eight directions by the names of the points on a compass north, northeast, east, southeast, south, southwest, west, and northwest, or N, NE, E, SE, S, SW, W, NW.



We must be careful here because not every position has eight neighbors. If (i,j) is on a border where either i = 1 or m, or j = 1 or n, then less than eight and possibly only three neighbors exist. To avoid checking for these border conditions we can surround the maze by a border of ones. The array will therefore be declared as MAZE(0:m + 1,0:n + 1).
Another device which will simplify the problem is to predefine the possible directions to move in a table, MOVE(1:8.1:2), which has the values

MOVE	1	2	
	--	--	
(1)	-1	0	north
(2)	-1	1	northeast
(3)	0	1	east
(4)	1	1	southeast
(5)	1	0	south
(6)	1	-1	southwest
(7)	0	-1	west
(8)	-1	-1	northwest

By equating the compass names with the numbers 1,2, ...,8 we make it easy to move in any direction. If we are at position (i,j) in the maze and we want to find the position (g,h) which is southwest of i,j, then we set

g i + MOVE(6,1); h j + MOVE(6,2)

For example, if we are at position (3,4), then position (3 + 1 = 4, 4 +(-1) = 3) is southwest.
As we move through the maze we may have the chance to go in several directions. Not knowing which one to choose, we pick one but save our current position and the direction of the last move in a list. This way if we have taken a false path we can return and try another direction. With each new location we will examine the possibilities, starting from the north and looking clockwise. Finally, in order to prevent us from going down the same path twice we use another array MARK(0:m + 1,0:n + 1) which is initially zero. MARK(i,j) is set to 1 once we arrive at that position. We assume MAZE(m,n) = 0 as otherwise there is no path to the exit. We are now ready to write a first pass at an algorithm.

```
set list to the maze entrance coordinates and direction north;
while list is not empty do
(i,j, mov)  coordinates and direction from front of list
while there are more moves do
(g,h)  coordinates of next move
if (g,h) = (m,n) then success
if MAZE (g,h) = 0        //the move is legal//
and MARK (g,h) = 0       //we haven't been here before//
then [MARK (g,h)  1
add (i,j, mov) to front of list
(i,j,mov)  (g,h, null)]
end
end
print no path has been found
```

This is not a SPARKS program and yet it describes the essential processing without too much detail. The use of indentation for delineating important blocks of code plus the use of SPARKS key words make the looping and conditional tests transparent.
What remains to be pinned down? Using the three arrays MAZE, MARK and MOVE we need only specify how to represent the list of new triples. Since the algorithm calls for removing first the most recently entered triple, this list should be a stack. We can use the sequential representation we saw before. All we need to know now is a reasonable bound on the size of this stack. Since each position in the maze is visited at most once, at most mn elements can be placed into the stack. Thus mn locations is a safe but somewhat conservative bound. In the following maze the only path has at most m/2 (n + 1) positions. Thus mn is not too crude a bound. We are now ready to give a precise maze algorithm.

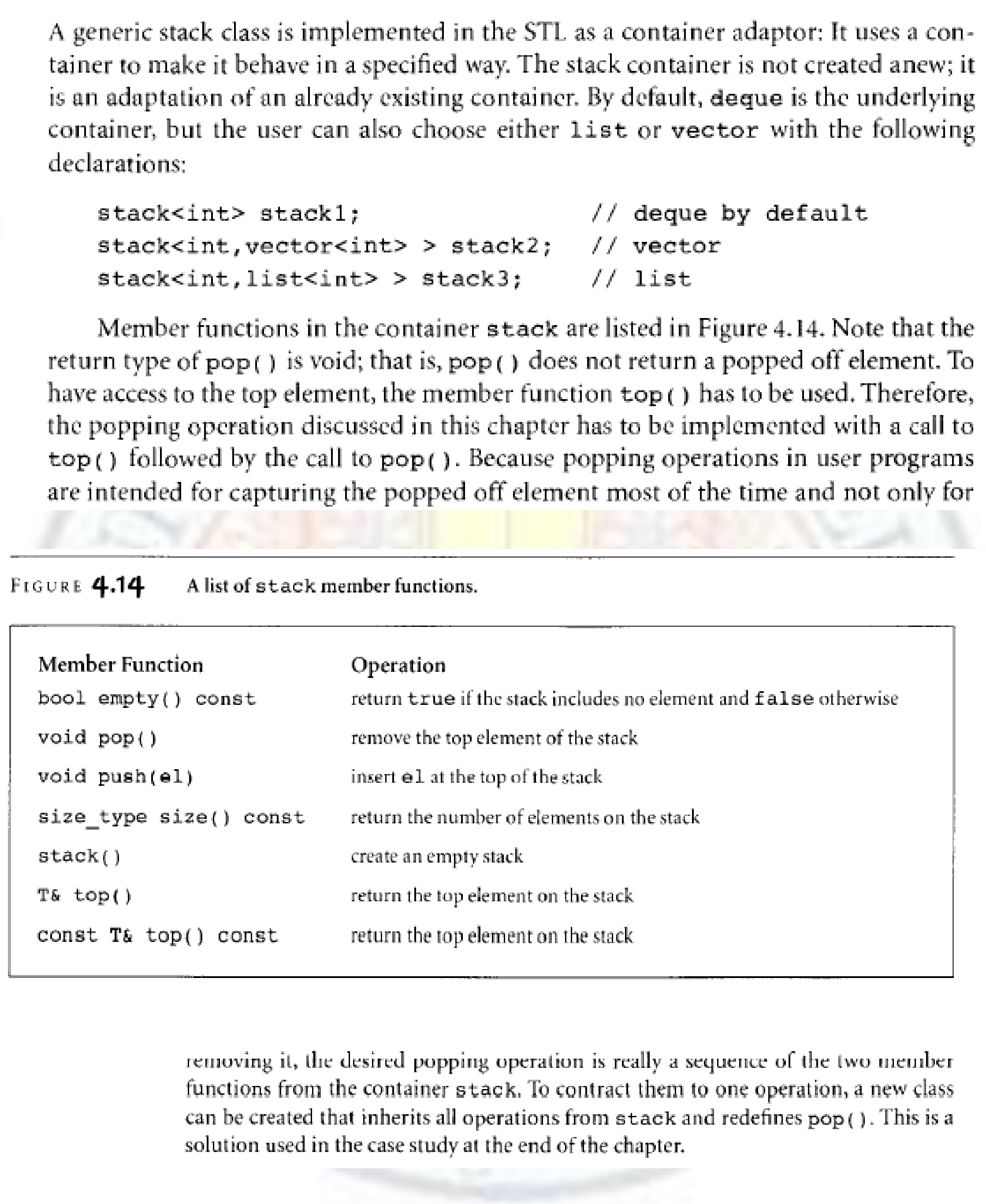
```
procedure PATH (MAZE, MARK, m, n, MOVE, STACK)
//A binary matrix MAZE (0:m + 1, 0:n + 1) holds the maze.
```

```
MARK (0:m + 1, 0:n + 1) is zero in spot (i,j) if MAZE (i,j) has not
yet been reached. MOVE (8,2) is a table used to change
coordinates
(i,j) to one of 8 possible directions. STACK (mn,3) holds the
current path// MARK (1,1)  1
(STACK(1,1),STACK(1,2),STACK(1,3))  (1,1,2);top  1

while top 0 do
(i,j,mov)  (STACK(top,1),STACK(top,2), STACK(top,3) + 1)
top  top - 1
while mov  8 do
g  i + MOVE (mov,1); h  j + MOVE(mov,2)
if g = m and h = n
then [for p  1 to top do     //goal//
print (STACK(p,1),STACK(p,2)
end
print(i,j); print(m,n);return]
if MAZE(g,h) = 0 and MARK(g,h) = 0
then[MARK(g,h)  1
top  top + 1
(STACK(top,1),STACK(top,2),STACK(top,3))
(i,j,mov)    //save (i,j) as part of current path//
mov  0; i  g; j  h]
mov  mov + 1          //point to next direction//
end
end
print ('no path has been found')
end PATH
```

Now, what can we say about the computing time for this algorithm? It is interesting that even though the problem is easy to grasp, it is difficult to make any but the most trivial statement about the computing time. The reason for this is because the number of iterations of the main while loop is entirely dependent upon the given maze. What we can say is that each new position (i,j) that is visited gets marked, so paths are never taken twice. There are at most eight iterations of the inner while loop for each marked position. Each iteration of the inner while loop takes a fixed amount of time, O(1), and if the number of zeros in MAZE is z then at most z positions can get marked. Since z is bounded above by mn, the computing time is bounded by . (In actual experiments, however, the rat may be inspired by the watching psychologist and the invigorating odor from the cheese at the exit. It might reach its goal by examining far fewer paths than those examined by algorithm PATH. This may happen despite the fact that the rat has no pencil and only a very limited mental stack. It is difficult to incorporate the effect of the cheese odor and the cheering of the psychologists into a computer algorithm.) The array MARK can be eliminated altogether and MAZE(i,j) changed to 1 instead of setting MARK(i,j) to 1, but this will destroy the original maze.

Source: Fundamentals of Data Structures - Ellis Horowitz

Value addition: Do you Know
How is stack implemented using standard template library?



A generic stack class is implemented in the STL as a container adaptor: It uses a container to make it behave in a specified way. The stack container is not created anew; it is an adaptation of an already existing container. By default, `deque` is the underlying container, but the user can also choose either `list` or `vector` with the following declarations:

```
stack<int> stack1;                 // deque by default
stack<int,vector<int> > stack2;   // vector
stack<int,list<int> > stack3;     // list
```

Member functions in the container `stack` are listed in Figure 4.14. Note that the return type of `pop()` is void; that is, `pop()` does not return a popped off element. To have access to the top element, the member function `top()` has to be used. Therefore, the popping operation discussed in this chapter has to be implemented with a call to `top()` followed by the call to `pop()`. Because popping operations in user programs are intended for capturing the popped off element most of the time and not only for

FIGURE **4.14** A list of `stack` member functions.

Member Function	Operation
`bool empty() const`	return `true` if the stack includes no element and `false` otherwise
`void pop()`	remove the top element of the stack
`void push(el)`	insert `el` at the top of the stack
`size_type size() const`	return the number of elements on the stack
`stack()`	create an empty stack
`T& top()`	return the top element on the stack
`const T& top() const`	return the top element on the stack

removing it, the desired popping operation is really a sequence of the two member functions from the container `stack`. To contract them to one operation, a new class can be created that inherits all operations from `stack` and redefines `pop()`. This is a solution used in the case study at the end of the chapter.

Source:Data Structure And Algorithms In C++ 2nd ed - Adam Drozdek

Value addition: Misconception
Are Stack and Heap different?

.

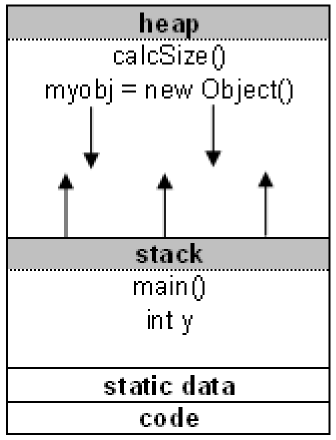
The **stack** is a place in the computer memory where all the variables that are declared and initialized *before* runtime are stored. The **heap** is the section of computer memory where all the variables created or initialized *at* runtime are stored.

What are the memory segments?

The distinction between *stack* and *heap* relates to programming. When you look at your computer memory, it is organized into three segments:

- text (code) segment
- stack segment
- heap segment

The **text segment** (often called **code segment**) is where the compiled code of the program itself resides. When you open some EXE file in Notepad, you can see that it includes a lot of "Gibberish" language, something that is not readable to human. It is the machine code, the computer representation of the program instructions. This includes all user defined as well as system functions.



What is stack**?**

The two sections other from the code segment in the memory are used for **data**. The **stack** is the section of memory that is allocated for **automatic variables within functions**.

Data is stored in stack using the Last In First Out (LIFO) method. This means that storage in the memory is allocated and deallocated at only one end of the memory called the **top of the stack.** Stack is a section of memory and its associated registers that is used for temporary storage of information in which the most recently stored item is the first to be retrieved.

What is heap**?**

On the other hand, **heap** is an area of memory used for **dynamic memory allocation**. Blocks of memory are allocated and freed in this case in an arbitrary order. The pattern of allocation and size of blocks is not known until run time. Heap is usually being used by a program for many different purposes.

The stack is much faster than the heap but also smaller and more expensive

Heap and stack from programming perspective

Most object-oriented languages have some defined structure, and some come with so-called main() function. When a program begins running, the system calls the function main() which marks the entry point of the program. For example every C, C++, or C# program must have one function named main(). No other function in the program can be called main(). Before we start explaining, let's take a look at the following example:

```
int x;                    /* static stack storage */
void main() {
   int y;                 /* dynamic stack storage */
   char str;              /* dynamic stack storage */
   str = malloc(50);      /* allocates 50 bytes of dynamic heap storage */
   size = calcSize(10);      /* dynamic heap storage */
```

When a program begins executing in the main() function, *all variables declared within main() will be stored on the stack*.

If the main() function calls another function in the program, for example calcSize(), additional storage will be allocated for the variables in calcSize(). This storage will be allocated in the **heap** memory segment.

Notice that the parameters passed by main() to calcSize() are also stored on the stack. If the calcSize() function calls to any additional functions, more space would be allocated at the heap again.

When the calcSize() function returns the value, the space for its local variables at heap is then deallocated and heap clears to be available for other functions.

The memory allocated in the heap area is used and reused during program execution.

It should be noted that memory allocated in heap will contain garbage values left over from previous usage.

Memory space for **objects** is always allocated in heap. Objects are placed on the heap.

Built-in datatypes like int, double, float and parameters to methods are allocated on the stack.

Even though objects are held on heap, references to them are also variables and they are placed on stack.

The stack segment provides more stable storage of data for a program. The memory allocated in the stack remains in existence for the duration of a program. This is good for global and static variables. Therefore, global variables and static variables are allocated on the stack.

Why is stack and heap important?

When a program is loaded into memory, it takes some memory management to organize the process. If memory management was not present in your computer memory, programs would clash with each other leaving the computer non-functional.

Source:http://www.maxi-pedia.com/what+is+heap+and+stack

Value addition: Do you Know

What are the disadvantages of stack ?

Buffer overflows, both on the stack and on the heap, are a major source of security vulnerabilities in C, Objective-C, and C++ code. This article discusses coding practices that will avoid buffer overflow problems, lists tools you can use to detect buffer overflows, and provides samples illustrating safe code. This article assumes familiarity with the concepts of memory allocation and the program's heap and stack. For a higher-level discussion of the problem, see “Buffer Overflows.”

The Source Of the Problem

Local variables are allocated on the stack, along with parameters and linkage information (that is, where to resume execution after a function returns.) The exact content and order of data on the stack depends on the operating system and CPU architecture. When you use malloc, new, or equivalent functions to allocate a block of memory or instantiate an object, the memory is allocated on the heap.

Every time your program solicits input from a user, there is a potential for the user to enter inappropriate data. For example, they might enter more data than you have reserved room for in memory. If the user enters more data than will fit in the reserved space, and you do not truncate it, then that data will overwrite other data in memory. If the memory overwritten contained data essential to the operation of the program, this overflow will cause a bug that, being intermittent, might be very hard to find. If the overwritten data includes the address of other code to be executed and the user has done this deliberately, the user can point to malicious code that your program will then execute.

In the case of data saved on the stack, such as a local variable, it is relatively easy for an attacker to overwrite the linkage information in order to execute malicious code. An attacker can also modify local data and function parameters on the stack. Figure 1 illustrates a stack overflow in Mac OS X running on a PowerPC processor. For other processors, the details are different, but the effect is the same.

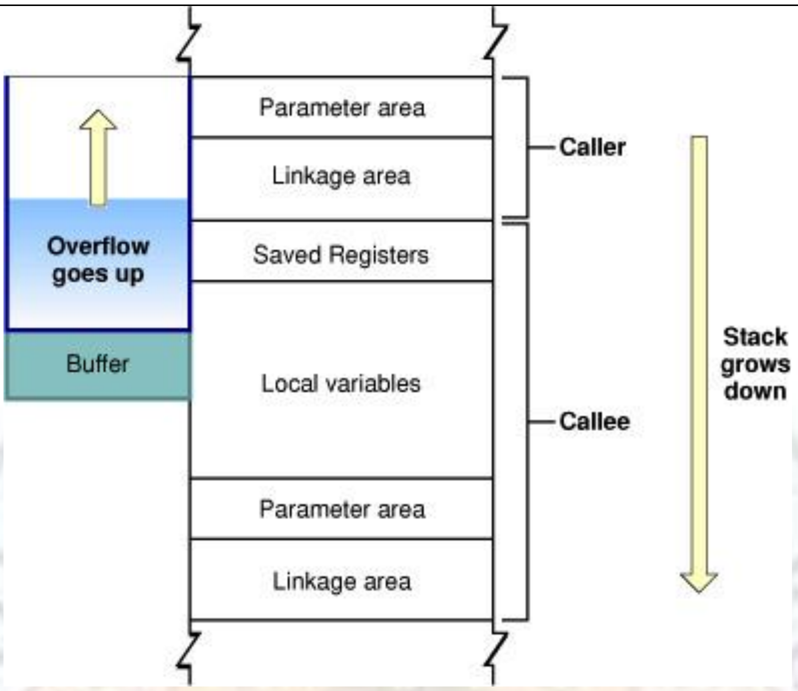


Figure 1 Mac OS X PPC stack overflow

Because the data on the heap changes in a nonobvious way as a program runs, exploiting a buffer overflow on the heap is more challenging. However, many successful exploits have involved heap overflows. Attacks on the heap might involve overwriting critical data, either to cause the program to crash, or to change a value that can be exploited later (such as when a program temporarily stores a user name and password on the heap and an attacker manages to change them). In some cases, the heap contains pointers to executable code, so that by overwriting such a pointer an attacker can execute malicious code. Figure 2 illustrates a heap overflow overwriting a pointer.

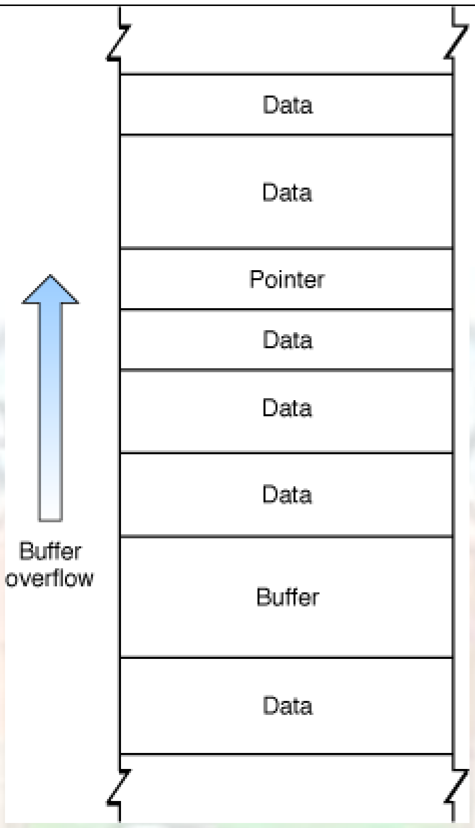


Figure 2 Heap overflow

Although most programming languages check input against storage to prevent buffer overflows, C, Objective-C, and C++ do not. Because many programs link to C libraries, vulnerabilities in standard libraries can cause vulnerabilities even in programs written in "safe" languages. For this reason, even if you are confident that your code is free of buffer overflow problems, you should limit exposure by running with the least privileges possible. Keep in mind that obvious forms of input, such as strings entered through dialog boxes, are not the only potential source of malicious input. For example:

1. Buffer overflows in one operating system’s help system could be caused by maliciously prepared embedded images.

2. A commonly-used media player failed to validate a specific type of audio files, allowing an attacker to execute arbitrary code by causing a buffer overflow with a carefully crafted audio file.

Source:**http://developer.apple.com/iphone/library/documentation/Security/Conceptual/ SecureCodingGuide/Articles/BufferOverflows.html**

Summary

- In this chapter, we have studied that Stack is an abstract data structure which operates through only end called Top.
- It is used for both insertion and deletion of an element. It works on a principle of Last in first out.
- Its main operations are Push and Pop for inserting and deleting an element respectively from the stack.. Insertion and deletion is allowed from the Top .It moves one pointer top to reflect the status of stack at that moment.
- Stack is implemented mainly using two representations : Array and Linked. Array implementation is static in nature but Linked representation involves dynamic nature of stack to grow.
- Stack can be represented as software and hardware stack. Hardware stack is the stack used in memory during processing of programs.
- Stack can be used in solving expressions, delimiter matching, recursion , reversing a string , adding two large numbers etc.
- Stack and heap are two different data structures used .
- We can use standard template library for the functions of stack.
- Stack has an Overflow problem whenever used for data storage.

Exercises

Q.1 Design a class stack and implement all the functions required to implement Stack?

Q.2 Write the code for solving Arithmetic Expression using stack?

Q. 3. Write the program to implement Linked stack Using Singly linked list and Doubly Linked List.

Q.4 Write the following functions using stack ADT

a) Copy content from one stack to another in same order.

b) Copy content from one stack to another in reverse order.

Q.5 Write code for adding two large numbers using stack.

Q.6 Write the following functions using stack

a) check whether two stacks are identical

b) replace all occurrences of element with the new element.

c) count the number of elements in stack.

Q.7 Write the code for matching Delimiters in a given text .

Glossary

Abstract data type

A data type whose properties (domain and operations) are specified independently of any particular representation : a class of data objects with a defined set of operations that process the data objects while maintaining its properties.

Algorithm

A logical sequence of discrete steps that describes a complete solution to a given problem.

Big-O notation

A notation that expresses computing time (complexity) as the term in a function that increases most rapidly relative to the size of the problem.

Linked List

A list in which every node has a pointer which points to the next element of the list.

Constructor

An operation that builds new instances of an abstract data type .

Delimiter

A symbol or keyword that marls the beginning or end of the construct.

Doubly Linked list

A linked list in which each node is linked to both its successor and its predecessor.

Dynamic data structure

A data structure that can expand and contract during program execution.

Implementing

Coding and testing an algorithmss

Index

A value that selects the component from the array.

Overflow

The condition that arises when the value of the calculation is too large to be represented.

stack

a stack is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations: *push* and *pop*

References

Suggested Readings

1. Fundamentals of Data Structures - Ellis Horowitz
2. Data Structure And Algorithms In C++ 2nd ed - Adam Drozdek
3. Algorithms and Data Structures in CPlusPlus - Alan Parker
4. C++ plus Data Structures 4th ed - Nell Dale
5. Data Structures and Algorithms - Alfred V. Aho
6. Data Structures and Program Design in C++ - Robert L. Kruse
7. Data Structure using C and C++ - Langsam ,Augenstein and Tanenebaum

Web Links

1. www.codefords.wordpress.com
2. www.Wikipedia .org
3. www. read.cs.ucla.edu
4. www.cs.usfca.edu