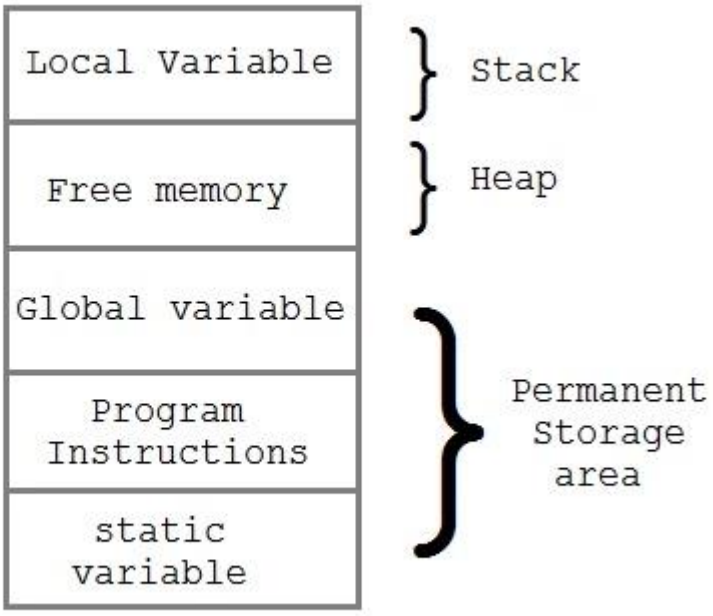
Dynamic Memory Allocation

Some programming languages allocate memory to the variables at the **compilation time**. This is called **static memory allocation**. This requires that all the variables used in the program be completely defined in the source program. For variables like arrays the programmer has to estimate the requirement which may result in shortage or wastage of memory.

C language provides facilities to allocate memory to variables during the **execution time** and also to release memory when no longer required. This is called **dynamic memory allocation**.

In C language there are four functions used with dynamic memory management. – ***malloc()***, ***calloc()*** and ***realloc()*** for allocation and ***free()*** *to return memory*. These functions are defined in the header file <stdlib.h>.

A conceptual view of memory is shown below though the actual implementations may vary slightly.



main() must be in memory all the times. **Called functions** may be present in the memory only when active or at all times (depending on the implementation)
Local variables for a function available only when the function is active and are managed from the **stack**.
Dynamic memory allocation is done using the **heap**.

malloc()

- Allocates a block of memory as specified in its parameters.
- Returns a pointer to the first byte of the allocated memory.
- The allocated memory is not initialized (contains garbage).

Prototype:

void * malloc(size_t *size*);

- returns a pointer to *size* bytes of memory block else returns NULL.
- *size* is usually specified using sizeof() function.

Usage:

ptr = malloc (sizeof(data_type));

It is better to make the pointer to be of a specific type using typecasting as below:

ptr = (data_type *) malloc (sizeof(data_type));

E.g.:
int *p;
p = (int *) malloc(sizeof(int)); // allocates two bytes of memory to store an integer

calloc()

- primarily used to allocate memory to arrays
- Allocates contiguous block of memory for the array
- Returns a pointer to the first byte of the allocated memory.
- The allocated memory is initialized (memory is cleared).

Prototype:

void *calloc(size_t *element_count*, size_t *element_size*);

- returns a pointer to *size* bytes of memory block else returns NULL.
- *size* is usually specified using sizeof() function.

Usage:

ptr = calloc (integer value, sizeof(data_type));

It is better to make the pointer to be of a specific type using typecasting as below:

ptr = (data_type *) calloc (count, sizeof(data_type));

E.g.:
int *p;
p = (int *) calloc(200, sizeof(int)); // allocates memory for 200 integers

realloc()

- changes the size of the previously allocated memory block.
- Can be highly inefficient and hence should be used carefully.
- **Prototype**:

void *realloc(void *ptr, size_t newsize);

E.g.:

```
int *p;
p = (int *) malloc(sizeof(int));    //initially only 2 bytes allocated
p = (int *) realloc(p, 15 * sizeof(int));    // reallocates memory for 15 integers
```

free()

- When memory allocated by malloc(), calloc() or realloc() are no longer needed, it can be freed.
- **Prototype**:

void free(void *ptr);

E.g.:

free(ptr);

Note: Any pointer to the freed memory is not destroyed and hence it is better to set it to NULL and avoid errors.

Programming example:

//Write a program that uses a table of integers whose size will be specified interactively at run time.

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
void main()
{
   int *p, *table;
   int size;
   clrscr();
   printf("\nWhat is the size of the table");
   scanf("%d",&size);
   table = (int *)malloc(size * size(int)); // allocates memory for specified numbers using malloc
   printf("Enter %d numbers", size);
   for(p=table; p<table + size; p++)
        scanf("%d",p);
  for(p=table; p<table + size; p++)
       printf("%d is stored at address %u",*p, p);
   getch();
}
```

//Same program using calloc function

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
void main()
{
   int *p, *table;
   int size;
   clrscr();
   printf("\nWhat is the size of the table");
   scanf("%d",&size);
   table = (int *)calloc(size, size(int)); // allocates memory for specified numbers using calloc
   printf("Enter %d numbers", size);
   for(p=table; p<table + size; p++)
        scanf("%d",p);
  for(p=table; p<table + size; p++)
       printf("%d is stored at address %u",*p, p);
   getch();
}
```

Data Structures

What is a Data Structure?
A **data structure** is a particular way of and organizing data in a computer so that it can be retrieved and manipulated efficiently.

Data structures are required to represent the real world relationships of data and for efficient processing. Usually, efficient data structures are a key to designing efficient algorithms. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. Examples of data structures include Arrays, Stacks, Queues, Linked Lists, Trees and Hash Tables

Note: *Data type* refers to the kinds of data that variables may 'hold', in a programming language. *Data object* refers to the set of elements. *Data structure* refers to the data objects and also the set of operations that may be applied to the elements of data object.

Classification of Data Structures

Two basic categories
1. Primitive
2. Non-primitive

1. Primitive data structures: These are basic data types which are directly operated upon by machine-level instructions.
E.g. Integers, floating-point numbers, characters, pointers.
2. Non-primitive data structures: These are user defined data types which are derived from the primitive data types.
Non primitive data structures are further classified into two categories
I. Linear data structures
II. Non Linear data structures

I. Linear data Structures: In this type elements have logical adjacency relationship i.e. elements are stored and accessed linearly(sequential order).
E.g. **Arrays, Stacks, Queues, Linked Lists**

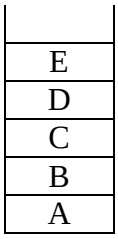
II. Non Linear Data Structures
in this a data item is connected to several other data items. So that a given data item has the possibility to reach one-or-more data items.
E.g.: **Graphs, Trees.**

Arrays

It is the simplest non-primitive data structure. An array is an ordered set whose elements are stored in consecutive memory locations. It contains a fixed number of objects. No deletions or insertions are performed on arrays.

Stack

A **stack** is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the **top** of the stack. The last element inserted is the first element to be taken out. Hence a stack is also called as **Last-In First-Out** (LIFO) list.



There are two operations that can be performed on a stack. When an item is added to the stack, it is called **push** operation, and when an item is removed from a stack it is called **pop** operation. Given a stack s and an item i, the operation **push(s,i)** adds the item i to the top of the stack s, and the operation **pop(s)** removes the top element and returns it as the function value e.g., i=pop(s). A stack is also referred to as a **pushdown list.**

Theoretically there is no limit on the number of elements that can be stored on a stack. An attempt to pop an element from an *empty stack*, is an illegal operation and results in an error condition called **underflow**. An operation empty(s) is defined which returns a TRUE if the stack is empty otherwise FALSE. Before popping an element from a stack, it is advisable to check if it is not empty using empty(s).

Problem: Verification of parentheses
A stack can be used to verify if the parentheses in the expression form an admissible pattern i.e.,
a. There are equal numbers of right and left parentheses.
b. Every right parenthesis is preceded by a matching left parenthesis.

Logic: The given expression is scanned from left end. Each left parenthesis is treated as opening a scope and every right parenthesis is treated as closing a scope. A *parentheses count* is defined as the number of left parenthesis minus the number of right parenthesis that have been encountered. Two conditions must hold:
1. The parentheses count at the end of the expression is 0. This implies that no scopes have been left open.
2. The parentheses count at each point in the expression is nonnegative. This implies that no right parenthesis is encountered for which a matching left parenthesis had not been previously encountered.

Applications of stack: Verification of scope delimiters
Let us assume that three types of scope delimiters are used in expressions viz. Parentheses – (), Braces- {} and Brackets []. It is desired to verify if a given expression is valid.

In the expression, *the last scope opened must be the first to be closed*. Hence A stack can be used to verify if the scope delimiters in the expression form an admissible pattern using the procedure below:
1. Whenever a scope opener is encountered, it is pushed on to the stack.
2. Whenever a scope ender is encountered, the stack is examined. If the stack is empty, the scope ender does not have a matching scope opener and hence the expression is invalid.
3. If, the stack is not empty, it is popped and the popped element is checked if it corresponds to the scope ender encountered most recently. If a match occurs the process is continued, else the expression is invalid.

4. When the end of the stack is reached, the stack must be empty; else one or more scopes have been opened but not closed and the string is invalid.

The algorithm for this procedure is given below:

```
valid =true;  // Assume the given expression is valid
s=the empty stack;
while(we have not read the entire string)
 { read the next symbol (symb) of the string;
   if(symb=='(' || symb=='{' || symb=='[' )
     push(s,symb);
   if(symb==')' || symb=='}' || symb==']' )
   if(empty(s))
       valid=false;
   else
     { i=pop(s);
       if(i is not the matching opener of symb)
       valid=false;
       } /* end else */
   } /* end while */

  if(!empty(s))
      valid=false;

    if(valid)
      printf("The String is valid");
    else
      printf("The String is invalid");
```

Representing stacks in C

The simplest implementation of a stack in C language can be done using an array and an integer variable. One end of the array is fixed as the bottom of the stack (normally the lowest index position - 0). Insertions and deletions are done at the other end. In order to indicate which position of the array is the current top of the stack, the integer variable is required. Hence a stack will be a structure. A stack of integers can be declared as below:

```
#define STACKSIZE 100
struct stack {
  int top;
  int items[STACKSIZE];
};
```

Now an actual stack **s** can be declared as:
struct stack s;

Whenever an element is inserted, *top* is first incremented by 1 and into that index position in the array, the element is stored. When the stack is popped, *top* is decremented by 1. Hence when the stack is empty the *top* is must have a value -1.

Whether a stack is empty or not can be found by checking if the value of *top* is -1 or not. A function **empty()** that returns a TRUE or FALSE can be written as below:

```
int empty(struct stack *ps)
{
   if (ps->top == -1)
       return (TRUE);
  else
     return(FALSE);
 }
```

Pop Operation

First, a check is made if the stack is empty. If so, error is indicated and the process is stopped. Else the element in the array at the index value = top is returned. Also, the top is decremented by one.

```
/* ps is a pointer to a stack of integers */
int pop(struct stack *ps)
{
  if (empty (ps))
    { printf("Stack Underflow");
      exit(1);
    } /* end if */
   return(ps->items[ps->top--]);
} /* end pop */
```

If the program execution must not halt upon the detection of underflow, the call to the function may be given as below:

```
if(!empty(&s))
  x=pop((&s);
else
  /* alternate action */
```

Now, the function pop() defined above need not check for empty stack and may have only the return statement

Push Operation

To push an element, the top is incremented by one and the element is stored into index position=top in the array

```
/* ps is a pointer to a stack of integers */
void push(struct stack *ps, int x)
{
  ps->items[++ps->top]=x);
  return;
} /* end push */
```

As the stack has been implemented using an array, its capacity is limited to the size of the array. Hence a stack can become full and an attempt to push one more element results in an error, termed **overflow**.

Hence the push() operation is revised as under:

```
/* ps is a pointer to a stack of integers */
void push(struct stack *ps, int x)
{
    if(ps->top == STACKSIZE-1)
      { printf("Stack overflow");
        exit(1);
        } /* end if */
   ps->items[++ps->top=x]);
   return;
 } /* end push */
```

stacktop(s)

This operation returns the top element of a stack without removing it from the stack. In fact it is not a basic operation and can be implemented as:

```
x=push(s);
push(s,x);
```

However a separate function can be written for this as under:

```
/* ps is a pointer to a stack of integers */
int stacktop(struct stack *ps)
{  if (empty (ps))
     { printf("Stack Underflow");
       exit(1);
       } /* end if */
    return(ps->items[ps->top]);
     } /* end stacktop */
```

Applications of Stack: Infix, Postfix and Prefix notations and evaluation of expressions.

Expressions for performing computations can be in one of the three forms below:

1. Operator is **in between** the operands – **Infix** notation. E.g. A + B
2. Operator **precedes** the operands – **Prefix** notation E.g. + A B
3. Operator is put **after** the operands – **Postfix** notation E.g. A B +

Infix expressions are evaluated applying the rules of operator precedence. If the precedence of operators has to be overridden parentheses are used. It is possible to convert an infix expression into prefix and postfix forms.

Infix	Prefix	Postfix
A+B	+AB	AB+
A+B-C	-+ABC	AB+C-
(A+B)*(C-D)	* +AB-CD	AB+CD-*
A$B*C-D+E/F/(G+H)	+-*$ABCD/ /EF+GH	AB$C*D- EF/GH+/+
((A+B)*C-(D-E))$(F+G)	$-*+ABC-DE+FG	AB+C* DE--FG+$
A-B/(C*D$E)	-A/B*C$DE	ABCDE$*/-

Evaluating a postfix expression

In postfix notation, each operator refers to the previous two operands. Hence evaluating an expression can be done easily using a stack. The expression is scanned one character at a time from left end. When an operand is read, it is pushed on to the stack. When an operator is read, the top two elements on the stack (which are the operands for that operator) are popped. The operation is performed and the result is pushed on to the stack. The scanning of the expression is continued further. After the expression is completely scanned, the result of evaluation will be present in the stack.

```
opndstk= empty stack;
/* scan the input string reading one character at a time into symb */
while(not end-of-input)
{  symb=next input character;
   if(symb is an operand)
     push(opndstk, symb);
   else
     { /*symb is an operator */
       opnd2=pop(opndstk);
       opnd1=pop(opndstk);
       value=result of applying symb to opnd1 and opnd2
       push(opndstk, value);
       } /* end else */
    } /* end while */
return(pop(opndstk));
```

Trace of the above algorithm for evaluating the expression 382/+2$3+ is given below. Contents of symb, opnd1, opnd2, value and the opndstk are shown.

symb	opnd1	opnd2	value	opndstk
3				3
8				3,8
2				3,8,2
/	8	2	4	3,4
+	3	4	7	7
2				7,2
$	7	2	49	49
3				49,3
+	49	3	52	52

Converting an expression from Infix to Postfix notation

Precedence of operators has to be considered for converting an expression from infix to postfix notation. The infix expression is scanned from the beginning character by character. If the scanned symbol is an operand it is added to the postfix string. A stack – opstk, is used to 'remember' the operators encountered. If the scanned symbol is an operator, and has precedence equal to or greater than the one on the top of the stack, it is pushed on to the stack. Else, the operator on the top of the stack has to be executed first and hence to be posted to the postfix string. The next operator on the new top of the stack also has to be compared with the scanned operator and process repeated. For this purpose, a function **prcd(op1,op2)** where op1 and op2 are operators is designed. The function returns TRUE if op1 has equal or greater precedence over op2 or a FALSE otherwise. Special case is: prcd('$', '$') is FALSE.

Algorithm:

```
opstk = empty stack;
while(not end of input)
{ symb= next input character;
   if(symb is an operand)
     add symb to the postfix string;
   else
    { while(!empty(opstk) && prcd(stacktop(opstk),symb)
       { topsymb = pop(opstk);
         add topsymb to the postfix string;
         } /* end while */
       push(opstk,symb);
       } /* end else*/
while(!empty(opstk))
{  topsymb = pop(opstk);
   add topsymb to the postfix string;
    } /* end while */
```

Trace of the above algorithm for converting the expression A+B*C to postfix is given below. Contents of symb, postfix string and opstk are shown.

symb	postfix string	Opstk
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *
	ABC*	+
	ABC*+	

When parentheses are present in the infix expression the following steps need to be included. When an opening parenthesis is read, must be pushed on to the stack. This can be done by defining **prcd(op, '(')=FALSE for any operator op other than ')'**. Also **prcd('(',op)=FALSE for any operator op**. This ensures that an operator symbol appearing after '(' is pushed on to the stack.

When a closing parenthesis is read, all operators up to the previous opening parenthesis must be popped and added to the postfix string. This can be by defining **prcd(op, ')')=TRUE for all operators other than '('.** After this, the '(' must be popped off the stack and not added to the postfix string. This can be done by defining **prcd('(', ')')=FALSE.** Also, the ')' must be discarded. For this, the push statement in the above algorithm is replaced by the if statement below:

```
if(empty(opstk) || symb != ')')
  push(opstk,symb);
else /* pop the opening parenthesis and discard it */
  topsymb=pop(opstk);
```

Trace of the modified algorithm for converting the expression (A+B)*C to postfix is given below. Contents of symb, postfix string and opstk are shown.

symb	postfix string	Opstk
(		(
A	A	(
+	A	(+
B	AB	(+
)	AB+	
*	AB+	*
C	AB+C	*
	AB+C*	

Recursion

Many objects in mathematics are defined by presenting a process (a formula) to produce that object.

Eg.: Factorial of a positive integer n is defined as the product of all integers from 1 to n.

It is written as:

$n! = 1$ if $n=0$;

$n! = n * (n-1) * \ldots * 3 * 2 * 1$ if $n>0$

The definition of factorial above is not a formula but a short hand notation for a process. We can define a function for the same as an algorithm below:

Eg:

Iterative function

```
fact = 1;
for(i=n; i >0 ; i--)
   fact *= i;
return(fact);
```

An algorithm where some process is repeated explicitly till a condition is met is called an **iterative** algorithm.

Factorial of a number can also be defined as below:

$n! = 1$ if $n=0$;

$n! = n * (n-1)!$ if $n>0$

Here the factorial of a positive integer is defined in terms of factorial of a number one less than it.

Such a definition where an object is defined *in terms of a simpler case of itself* is called a **recursive** definition.

An algorithm for implementing the above definition is given below:

```
fact(n)
{
  if n==0)
    return(1);
  else
     return(n*fact(n-1));
  } /* end fact */
```

A function which calls itself is called a **recursive function**. The function fact() calls itself with a parameter value 1 less, repeatedly. When the parameter value becomes 0, that call of the function returns a value 1. This value is returned to the previous call of the same function when the value of the previous call is computed and that value is returned to its previous call and so on. Finally the first call of the function gets the value of (n-1)! which is multiplied with n and the final value is determined.

Some processes are defined more easily using recursion.

Some more examples of recursive definitions are given below:

1. **Multiplication of natural numbers**: Product of two natural numbers a*b is given below:
   $a * b = a$ if $b=1$
   $a * b = a * (b-1) + a$ if $b>1$

2. **Fibonacci Sequence**: Fibonacci sequence is a sequence of integers, where the first and the second values are 0 and 1 and the successive values are obtained as the sum of previous two values. The series is: 0,1,1,2,3,5,8,13,21,34,55….. It is defined as:
fib(n) = n if n==0 or n==1
fib(n) = fib(n-2) + fib(n+1) if n>=2

Properties of recursive definitions or algorithms
1. A recursive algorithm should not generate infinite sequence of calls on itself. The simplest case of the term to be defined is defined explicitly: Non-recursive exit
2. The other cases are defined by applying some operation to the result of evaluating a simpler case.

Note: Sometimes there will be great deal of computational redundancy in recursive definitions. E.g. In Fibonacci sequence the same fibonacci number is computed many times.

Binary Search
If a list is unordered, sequential/ linear search is the only way to find anything in it. However, if the list is ordered, the maximum number of comparisons required to find an entry can be greatly reduced by using binary search. The concept of binary search naturally tends to be recursive. Depending on the value of the middle element (greater or smaller than the element being searched) a sublist (lower half or upper half) is searched. The searching procedure calls itself with a smaller sublist everytime.

Recursive algorithm for binary search:

```
binsearch(low, high, x)
 {
   if(low > high)
     return (-1)
   mid=(low+high)/2;
   if(x==a[mid]
     return a[mid];
   if(x < a[mid]
     binsearch(low, mid+1, x)
   else
     binsearch(mid+1, high, x)
   } /* end binsearch */
```

Recursion in C
C language allows the use of functions that call themselves – recursive functions. A recursive function to compute n! is given below:

```
int fact (int n)
{ int x,y;
  if(n==0)
    return (1);
  else
   {
      x=n-1;
      y= fact(x)
      return(n*y)
```

```
}
  } /* end fact */
```

The recursive function fact() above, calls itself, every time with value of n which is 1 less than the previous call. For each call of the function a new set of parameter **n** and the local variables are allocated. The last call to fact is with a value n=0, when the function returns a value 1. When a return to the previous call takes place the most recent allocation of variables is freed. This suggests that the successive generations of parameters and local variables be maintained on a stack. Snapshots of the stack during the execution of the function rfact() with n=4 is shown below:

n	x	y

n x y
Initial

n	x	y
4	*	*

n x y
fact(4)

n	x	y
3	*	*
4	3	*

n x y
fact(3)

n	x	y
2	*	*
3	2	*
4	3	*

n x y
fact(2)

n	x	y
1	*	*
2	1	*
3	2	*
4	3	*

n x y
fact(1)

n	x	y
0	*	*
1	0	*
2	1	*
3	2	*
4	3	*

n x y
fact(0)

n	x	y
1	0	1
2	1	*
3	2	*
4	3	*

n x y
y=fact(0)

n	x	y
2	1	1
3	2	*
4	3	*

n x y
y=fact(1)

n	x	y
3	2	2
4	3	*

n x y
fact(2)

n	x	y
4	3	6

n x y
fact(3)

n	x	y

n x y
fact(4)=n*y = 24

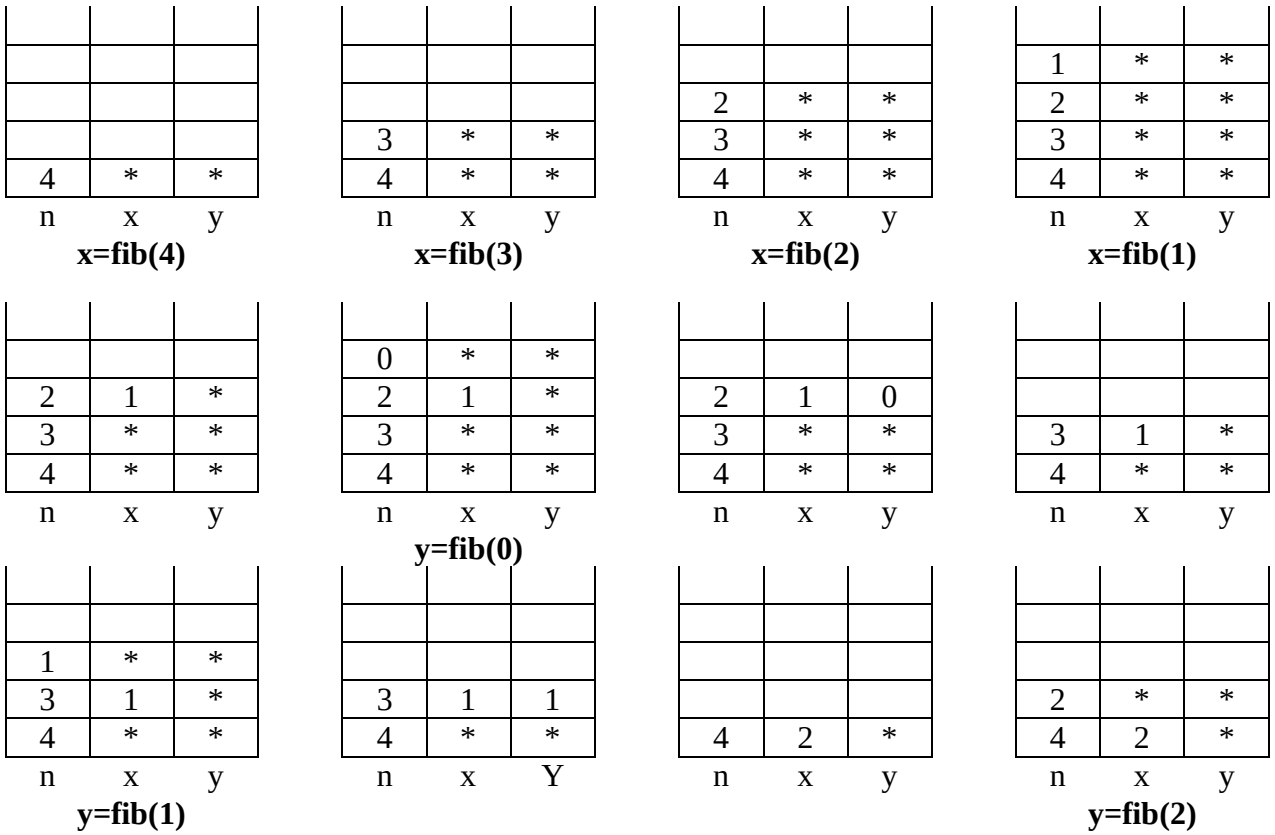
Recursive function to compute the product of two positive integers	**Compact version of the function**

int mult(int a, int b)
{ if(b==1)
return(a);
c=b-1;
d=mult(a,c)
return (a+d);
}/* end mult */
	int mult(int a, int b)
	{ return(b==1 ? a: mult(a,b-1)+a);
	}/* end mult */

C function to compute the Fibonacci sequence

```
int fib(int n)
{ int x,y;
  if(n<=1)
    return(n);
  x=fib(n-1);
  y=fib(n-2);
  return (x+y);
  }/* end fib */
```

Here also, successive generations of parameters and local variables are maintained on a stack. Snapshots of the stack during the computation of 4th Fibonacci number – fib(4), is shown below:



1	*	*
2	*	*
4	2	*

n x y

x=fib(1)

2	1	*
4	2	*

n x Y

0	*	*
2	1	*
4	2	*

n x y

y=fib(0)

2	1	0
4	2	*

n x y

4	2	1

n x y

The value (x+y)=3 is returned as the result of the call to fib(4)

The Towers of Hanoi Problem

Problem: Three pegs A, B, and C exist. Five disks of differing diameters are placed on peg A so that a larger disk is always below a smaller disk. The object is to move the five disks to peg C, using peg B as auxiliary. Only the top disk on any peg may be moved to any other peg, and a larger disk may never rest on a smaller one. :

Solution: Let us consider a general solution. If we have a solution for n-1 disks and could state a solution for n disks in terms of the solution for n-1 disks, the problem can be solved using a recursive procedure. The trivial case is of one disk and it can be moved from peg A to peg C. The recursive solution can be stated as below:

1. If n==1, move the single disk from A to C and stop.
2. Move the top n-1 disks from A to B using C as auxiliary.
3. Move the remaining disk from A to C.
4. Move the n-1 disks from B to C using A as auxiliary.

Input to the program is the number of disks **n**. The pegs may be named as A, B and C. But since we may move disks from any of these pegs to any other using the remaining as auxiliary, it is necessary to have three parameters: **frompeg**, **topeg** and **auxpeg**. The C program for the problem is given below:

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int n;
  void towers(int n,char from,char to,char aux);
  clrscr();
  printf("\nEnter the number of disks: ");
  scanf("%d",&n);
  towers(n,'A','C','B');
  getch();
}



void towers(int n,char from,char to,char aux)
{
```

```
 if(n>0)
        {
  towers(n-1,from,aux,to);
  printf("\n Move %d disk from %c to %c",n,from,to);
  towers(n-1,aux,to,from);
 }
 return;
}
```

The result for n=3 is given below:

```
This program solves the Towers of Hanoi problem
Input the number of disks: 3

A:Source    B:Destination       C:Auxiliary

Move disk 1 from peg A to Peg B
Move disk 2 from peg A to Peg C
Move disk 1 from peg B to Peg C
Move disk 3 from peg A to Peg B
Move disk 1 from peg C to Peg A
Move disk 2 from peg C to Peg B
Move disk 1 from peg A to Peg B

Total number of moves made is: 7
```