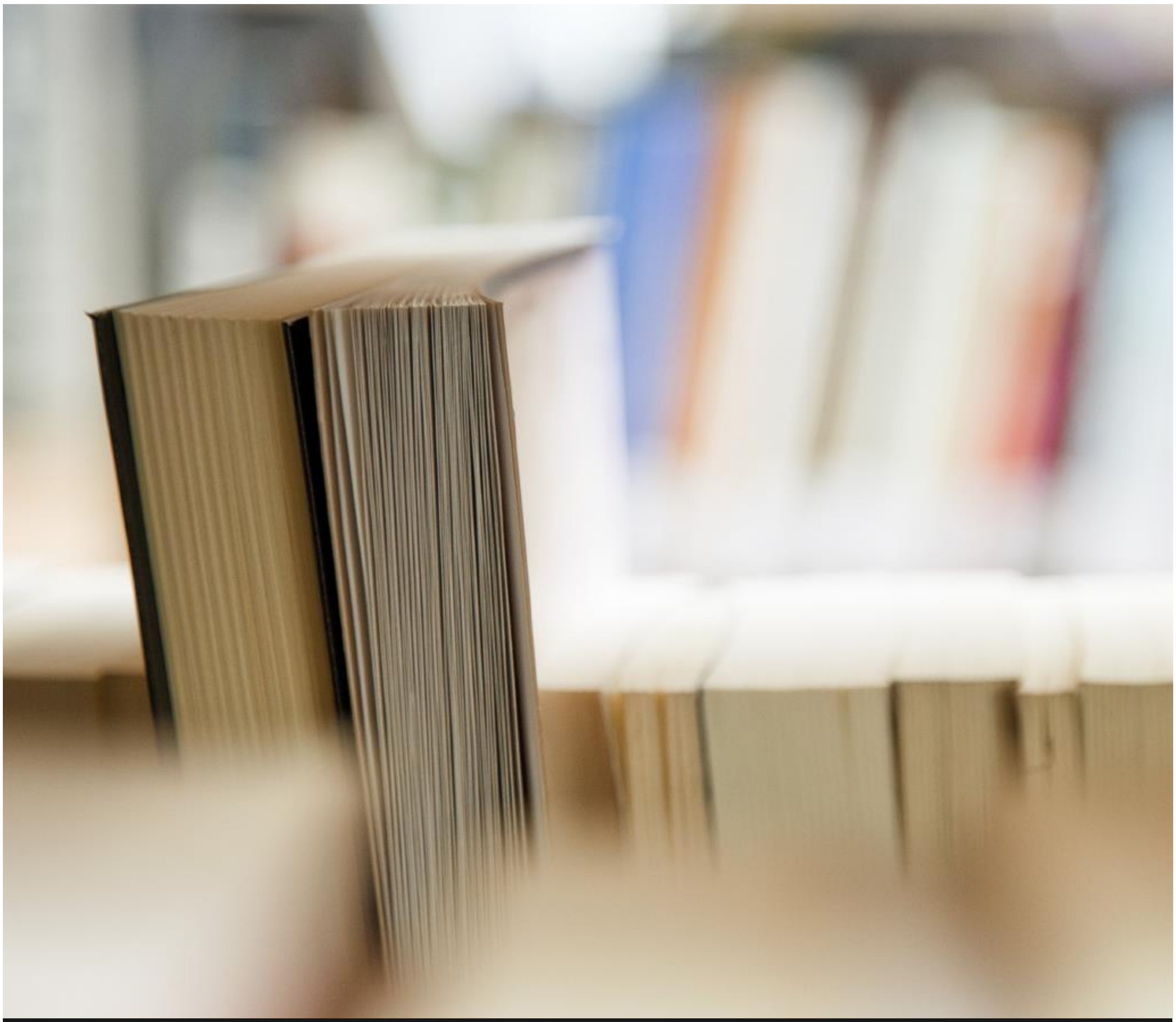


1. Introduction to Data Structures and Recursion

Data Structures and Algorithms [17ECSC204]

Prakash Hegade

2018-19

"If you can think, you can code!
If you can think better, you can code better!"
-PH

Chapter Contents:
C Basics, Arrays, Pointers, Structures and Operations, Recursion, Backtracking.

Introduction

The Ten Step Design Process

Following are the set of questions we need to ask while we design an algorithm:

1. What are all the legal inputs for the program?
2. What are all the possible outputs of the program?
3. How exactly is the output of the program related to the input?
4. What are the resources constraints for the program in terms of the allowed types of variables, guards and commands?
5. What are all the variables that are needed for the program?
6. How are the variables to be initialized?
7. How is the output of the program to be read?
8. How should the program change the values of the variables in one step?
9. Does the program terminate for every legal input?
10. If the program terminates for a legal input, is the output correct?

What is a computation?

Suppose we are asked to compute 9 * 6 and consider the following solutions:
Solution a. 9 * 6 = 54

Solution b. By referring to a 2-way multiplication table giving all products x* y and y between 2 and 10, we see that 9 * 6 =54

Solution c.

9	6	0
9	5	9
9	4	18
9	3	27
9	2	36
9	1	45
9	0	54

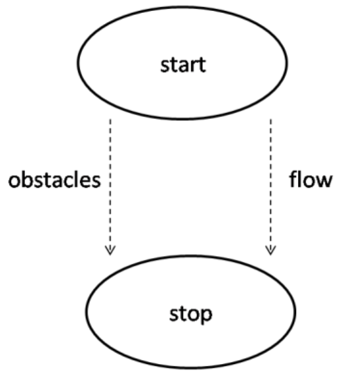
Which of the above three is a computation?

Solution a - There is no hint of how the solution was arrived, Cannot be a computation.
Solution b - Answers as special case of general question. An appropriate one was picked from a list of answers and hence cannot be a computation.
Solution c - 9 * 6 was converted to a tuple (9, 6, 0) and table seems to follow from the previous row by a definite rule. Multiplication has been treated as repeated addition. We now have a method that can be followed for any such calculations. This is a computation.

Properties of Computation:

1. It should be governed by a rule
2. The rule should be a general rule and work on all such similar problems
3. The rule should be clear and unambiguous
4. The process used in the rule must be at a greater level of detail than the problem itself.
5. There should be a stopping rule and when the computation stops we should be able to read the output.
6. The rule should be provably correct.

Design of New Algorithms



A new algorithm design can be tackled in 4 ways:
- Strength of start
- Strength of stop
- Strength in flow
- Winning over obstacles

One with a good start:

- Security algorithms – need to have strong assumptions
- “Integer factorization is hard” was the assumption made in discovery of ssh protocol

One with obstacles:

- Can we have a pool of algorithm and pick one at random?

- When obstacles are too difficult to beat, we go proactive – take small password as safe password is of infinite length and change it frequently
- Solving problems by introducing obstacles to obstacles. Example: rice price goes up → hold them in warehouses. Then to get the stock out, release the rats.

One with stop:

- We have a proper stop specification
- There are also problems with pseudo-randomness
- Approximate algorithms – change the stopping condition

One with the flow:

- Algorithm design techniques, which we shall study in detail through the course syllabus

On the whole:

- **S**tart well
- **O**bstacles guaranteed
- **F**low skillfully
- **T**erminate efficiently

And this is possibly the SOFT part of the software.

Pointer Basics

Let us start with a variable. The general syntax of the variable is given by:
data_type variable_name = value

Example,
int my_age = 24;
To print the value present in variable 'my_age' we can use,
printf("My Age is:%d\n", my_age);

Every variable is associated with a value and has an address. This means that we can refer to a variable by the name or the associated address. When we pick the address, it becomes the pointer variable.

How to get address of the variable *my_age*?
&my_age gives address of the variable.

We can store this address in another variable and that is how we get to the definition of a pointer variable.

Definition:

"A pointer is a variable which contains address of another variable"

Pointer Advantages:

- Support dynamic memory allocation
- Faster access of data
- We can access byte or word locations and the CPU registers directly
- The data in one function can be modified by other function by passing the address
- More than one value can be returned from a function through parameters using pointers
- Mainly useful while processing non-primitive data structures such as arrays, linked list etc

Pointer Disadvantages:

- Uninitialized pointers or pointers containing invalid address can cause the system to crash
- It is very easy to use pointers incorrectly, causing bugs that are very difficult to identify

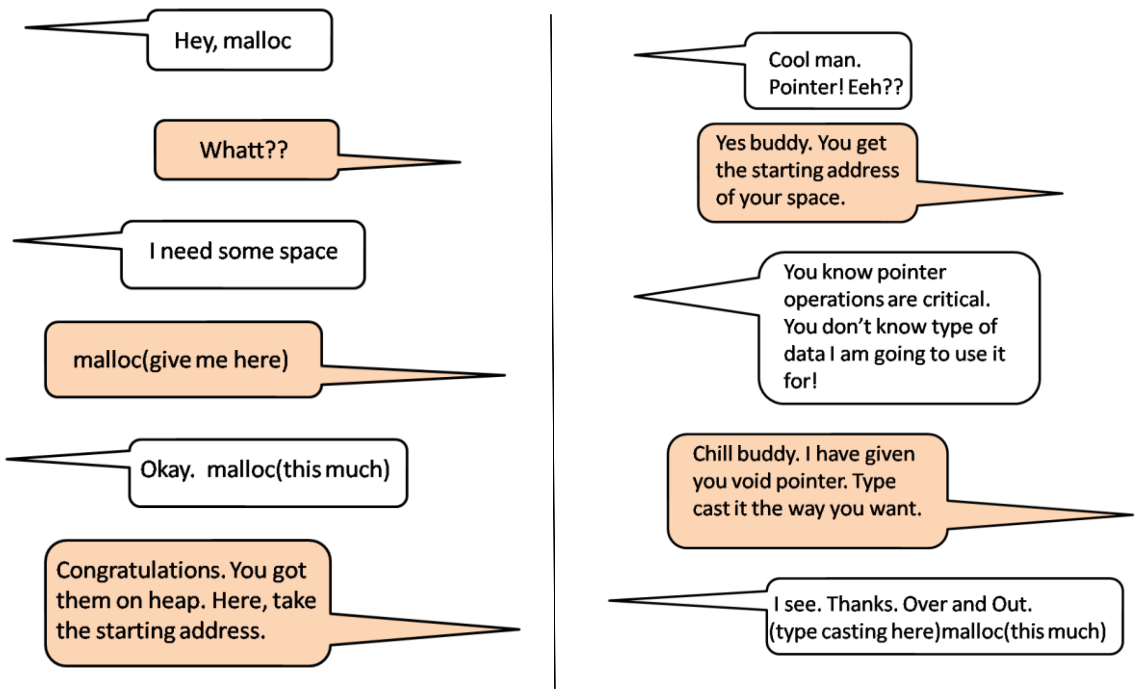
Dynamic Memory Allocation Using Malloc

Syntax:

ptr = (cast-type*) malloc(byte-size)

What will be the memory allocated for the following statement?
ptr = (int*) malloc(100 * sizeof(int));
Answer: 400 bytes assuming size of 'int' is 4 bytes

The working of malloc can be understood as explained in conversation figure below:


Hey, malloc
Whatt??
I need some space
malloc(give me here)
Okay. malloc(this much)
Congratulations. You got them on heap. Here, take the starting address.
Cool man. Pointer! Eeh??
Yes buddy. You get the starting address of your space.
You know pointer operations are critical. You don't know type of data I am going to use it for!
Chill buddy. I have given you void pointer. Type cast it the way you want.
I see. Thanks. Over and Out. (type casting here)malloc(this much)

Declaring and Initializing

How to distinguish a variable and a pointer variable??

A pointer variable is always prefixed with a '*'.

Syntax: *data_type * pointer_name;*

```
int number= 100;
int * p;
p = &number;
```

Note:

- int * p; *// p is a pointer variable which can hold the address of a variable of type int*
- double * q; *//q is a pointer variable which can hold the address of a variable of type double*
- Following are same: int *pa; int * pa; int * pa;
- When pointers are declared, it is better and safer to initialize it with NULL
- Global pointers are initialized to NULL during compilation

Program for accessing values using pointers

```
#include <stdio.h>
int main()
{
  int value, number=100;
  int * p;
  p = &number;
  value = *p;
  //100 can be obtained by all below statements:
  printf("%d\n", number);
  printf("%d\n", *p);
  printf("%d\n", value);
  printf("%d\n",*&number);
}
```

Address Arithmetic

Following operations are valid on Pointers:

- A pointer can be incremented or decremented
- Addition of integers to pointers is allowed
- Subtraction of two pointers of same data type is allowed
- The pointers can be compared using relational operators (p and q are two pointers)
    - p>q, p==q , p<=q, p<q
    - p==NULL, p!=q

Following operations are invalid on Pointers:

- Arithmetic operations such as addition, multiplication and division on two pointers are not allowed
- A pointer variable cannot be multiplied / divided by a constant or a variable
- Address of the variable cannot be altered. Example: &p = 2048

Pointers and Arrays

Compiler allocates memory to an array contiguously. The address of the array can be obtained by &a[0], &a[1] etc or also by specifying a, a+1 etc.
So, we can access address of ith element either by (a+i) or &a[i]

Note:

- Base address of array will be stored in the name given to the array
- Array name "**a**" can be considered as a pointer
- Array name "**a**" is pointer to the first element in the array
- Array and pointers go hand in hand

On base address "**a**",

- a++ is not valid
- a = &a[5] is not valid or any operation which modifies the address is not valid because we cannot change the base address. It is a **constant pointer**.

Program to Print Array Addresses

```
#include <stdio.h>
int main() {
  int array[4];
  int index;
  printf("The base address is %p\n", array);
  printf("All the array member addresses\n");
  for(index =0; index<4;index++) {
    printf("%p\n", array+index);
  }
  return 0;
}
```

Sample Output:

```
The base address is 0022FF0C
All the array member addresses
0022FF0C
0022FF10
0022FF14
0022FF18
```

Generalizing:
(array + index) = base address + index * number of bytes required to store an element

Note:
```
int array[4] = {1,2,3,4};
int *p;
p = array;        // p = &array[0];
```
Now the operations, p++, p = &a[2] are valid.

Program to print the array elements using pointers
```
#include <stdio.h>
int main()
{
   int array[] = {1,2,3,4};
   int index;
   int *p;
   p = array;

   // Method 01
   for(index =0; index<4;index++) {
      printf("%d\t", *p);
      p++;
   }

   // Method 02
   printf("\n");
   p = p - 4;
   for(index =0; index<4;index++) {
      printf("%d\t", *(p+index));
   }

   // Method 03
   printf("\n");
   for(index =0; index<4;index++) {
      printf("%d\t", index[p]);
   }

   // other valid statements: p[index], * (index+p)
   return 0;
}
```

Types of Pointers

1. **Pointer**: Pointer is a variable which holds the address of another variable

2. **NULL Pointer**: A null pointer has a value reserved for indicating that the pointer does not refer to a valid object. It is pointer initialized to NULL value

3. **Void Pointer**: are pointers pointing to data of no specific data type. The compiler will have no idea on what type of object the pointer is pointing to. It has to be type casted to the required type

4. **Dangling or Wild Pointer**: are pointers that do not point to a valid object of the appropriate type. A normal pointer becomes *dangling pointer* when it is free'd.

5. **Constant Pointer and Pointer to Constant**:
Consider an example:
char data = 'D';
char * p = &data;

const char * p -- declares a pointer to a constant character. We cannot use this pointer to change the value being pointed to.

char * const p -- declares a constant pointer to a character. The location stored in the pointer cannot change.

const char * const p -- declares a pointer to a character where both the pointer value and the value being pointed at will not change.

Other Pointer to be aware of: **Near** pointer, **Far** Pointer, **Huge** Pointer

Strings

- Array of characters
- Each String ends with a null character (\0) – only way for the string functions to know where the string will end
- If no '\0' it is not a string, it is collection of characters
- Defined in header <string.h>

Declaring and initializing:
char course[5] = {'D', 'S', '\0'};   OR
char course[5] = {"DS"};   OR
char course[5]= "DS";

Program to print a string using pointers

```
#include <stdio.h>
int main()
{
   char course[] = "DSA";
   char *p;
   p = course;
   while(*p != '\0') {
      printf("%c", *p);
      p++;
   }
   return 0;
}
```

All below are also valid:

course[i], i[course]
* (course + i), *(i + course)

String I/O functions:

To read:

- scanf( ) with %s format specification
- gets( ), getchar( )

To write:

- printf( ) with %s format specification
- puts( ), putchar( )

Note: To accept strings with space we generally use: gets(string); But using gets() is a bad programming practice.

Because it has the buffer limitation and can overwrite data even without any warnings. So a version of scanf() as specified below can be used to accept strings with space in it:
scanf("%[^\n]s", name);

Array of Strings:

Example:

char pens[3][20] = {“Reynolds”, “Parker”,”Cello” };
These strings can be accessed using only first subscript.
Example: printf(“%s”, pens[1]); will output Parker

String Manipulation Functions

1. strlen(str) – get the length of the string

```
#include <stdio.h>
int main()
{
   char str[30];
   int counter = 0;
   char *p = str;

   printf("Enter the string\n");
   scanf("%[^\n]s", str);
   while(*p != '\0') {
      counter++;
      p++;
   }
   printf("Length of the string is %d\n",
counter);
   return 0;
}
```

2. strcpy (str2, str1) - copies str1 to str2

```
#include <stdio.h>
int main()
{
   char str1[30];
   char str2[30];
   char *ptr1;
   char *ptr2;

   ptr1 = str1;
   ptr2 = str2;
   printf("Enter the String 1\n");
   scanf("%s", str1);
   while(*ptr1 != '\0') {
      *ptr2 = *ptr1;
      ptr1++;
      ptr2++;
   }
   *ptr2 = '\0';
   printf("The copied String is.. %s\n", str2);
   return 0;
}
```

3. strcat(str1, str2) - append str2 to str1

```
#include <stdio.h>
int main()
{
   char str1[40];
   char str2[20];

   char *ptr1 = str1;
   char *ptr2 = str2;

   printf("Enter two strings\n");
   scanf("%s %s", str1, str2);

  // Reach to the end of the str1
   while(*ptr1 != '\0')
      ptr1++;
   // Append the second string to first
   while(*ptr2 != '\0') {
      *ptr1 = *ptr2;
      ptr1++;
      ptr2++;
   }

*ptr1 = '\0';
 printf("The concatenated string is... %s", str1);

   return 0;
}
```

4. strcmp(str1, str2) - Compare two strings str1 and str2

```
#include <stdio.h>
int main()
{
   char str1[20];
   char str2[20];
   int result = 0;
   char *ptr1 = str1;
   char *ptr2 = str2;
   printf("Enter two strings\n");
   scanf("%s %s", str1, str2);

   while(*ptr1 == *ptr2)
   {
      if(*ptr1 == '\0')
         break;
      ptr1++;
      ptr2++;
   }

   result = *ptr1 - *ptr2;

   if(result ==0)
      printf("Two strings are equal\n");
   else if(result > 0)
      printf("First string is greater than second\n");
   else
      printf("Second string is greater than first\n");

   return 0;
}
```

5. strrev(str) - reverses all characters in the string

```
#include <stdio.h>

int main()
{
   char str[20];
   char *ptr = str;

   char *ptr1;
   char *ptr2;
   char temp;
   int length = 0;
   int index = 0;

   printf("Enter the string to be reversed\n");
   scanf("%s", str);

   while( *ptr != '\0')
   {
      length++;
      ptr++;
   }

   ptr1 = str;
   ptr2 = str + length-1;

   for(index = 0; index < length/2; index++)
   {
      temp = *ptr1;
      *ptr1 = *ptr2;
      *ptr2 = temp;
      ptr1++;
      ptr2--;
   }

   printf("Reversed String is... %s\n", str);
   return 0;
}
```

Case Example:

Check if a supplied string is a Palindrome

```
#include <stdio.h>
int main()
{
   char str[20];
```

```
    char *ptr = str;
    char *ptr1;
    char *ptr2;
    int flag = 0;
    int length = 0;
    int index = 0;

    printf("Enter the string\n");
    scanf("%s", str);
    while( *ptr != '\0')
    {
        length++;
        ptr++;
    }
    ptr1 = str;
    ptr2 = str + length-1;
    for(index = 0; index < length/2; index++)
    {
        if(*ptr1 == *ptr2)
        {
            ptr1++;
            ptr2--;
            continue;
        }
        else
        {
            flag = 1;
            break;
        }
    }
    if(flag == 0)
        printf("String is a Palindrome\n");
    else
        printf("String is not a Palindrome\n");
    return 0;
}
```

Other String Functions which you must know:

- **strncmp**: compares first n characters of two string inputs
    - Syntax: strncmp(str1, str2, n)
    - Returns : -1, 0 or 1

- **strncpy:** copies first n characters of the second string to the first string
    - Syntax: strncpy(str1, str2, n)
- **strncat:** appends first n characters of the second string at the end of first string.
    - Syntax: strncat(str1, str2, n)

- **strlwr(str) :** convert any uppercase letters in the string to the lowercase

- **strupr(str) :** convert any lowercase letters that appear in the input string to uppercase

- **strchr(str, char):** searches for specified character in the string. It returns NULL if the desired character is not found in the string.

Pointer Arrays

Since a pointer variable always contains an address, an array of pointers would be nothing but a collection of addresses.

The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses.

Can you differentiate between the following?

```
int a;          int a[10];          int *a;          int *a[10];
```

Program to demonstrate the usage of array of pointers

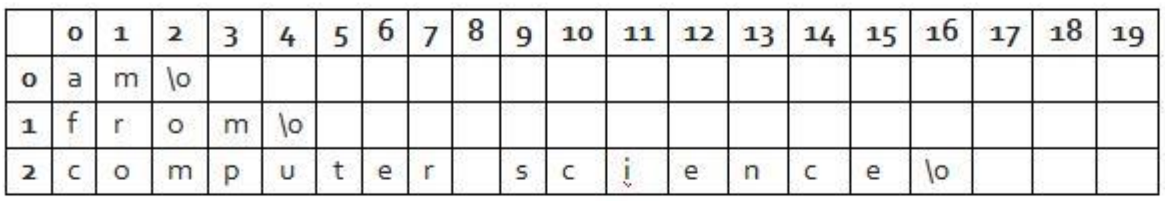
```
int main()
{
        int *arr[3];
        int i = 10, j = 20, k = 30;
        int index;
        arr[0] = &i;
        arr[1] = &j;
        arr[2] = &k;

        for(index = 0; index < 3; index++)
                printf("%d\n", *(arr[index]));

        return 0;
}
```

Operations Comparison Table:

	int array[i][j];	**int *array[i]**
	Array Indexing	Pointer Arithmetic
Address	&array[i][j]	array[i] + j
Value	array[i][j]	*(array[i]+ j)

Example of strings:

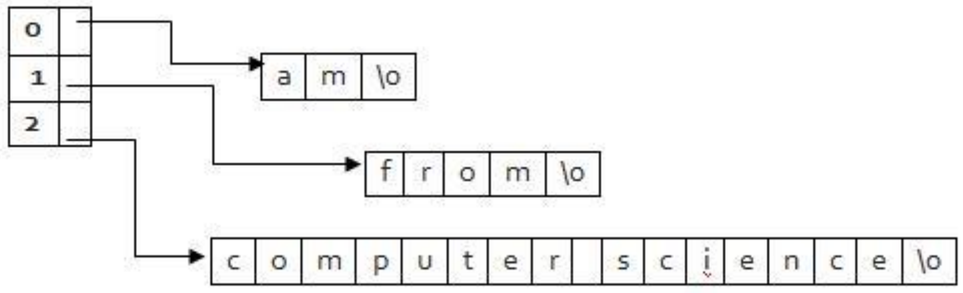
Similar concept can be extended to strings also. Consider for example,
A 2D- char array,
char a[3][20] = {“am”, “from”, “computer science”};



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	a	m	\o																	
1	f	r	o	m	\o															
2	c	o	m	p	u	t	e	r		s	c	i	e	n	c	e	\o			

The size of each row will be fixed during compilation and results in wastage of memory.

To avoid the wastage of memory, we can go for,
char *a[3] = {“am”, “from”, “computer science”};



Now **a** is an array of character pointers. Each location holds the address of respective allocated memory for representative string.

Note: Pointer Notations and Examples

int (*ptr) ( )

- Function Pointer
- Return type is integer
- No parameters passed to the function
- *ptr holds the address of the function (as good as name of the function)

int * ptr ( )

- ptr is a function
- No parameters passed to the function
- Return type of the function is int *
- Name of the function is ptr

***a[10]**

- array of pointers
- Each location pointing to specified data type

(*a)[10]

- a is a pointer to a group of contiguous 1-dimensional array elements

(*a)( )

- a is a function pointer
- Return type of the function is void
- No parameters passed to the function
- *a holds the address of the function

Iteration

Means the act of repeating a process with the aim of approaching a desired goal, target or result. Each repetition of the process is also called an "iteration," and the results of one iteration are used as the starting point for the next iteration.

Iteration in computing is the repetition of a block of statements within a computer program.

Recursion

A class of objects or methods exhibit recursive behaviour when they can be defined by two properties:

1. A simple base case (or cases)
2. A set of rules that reduce all other cases toward the base case

Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to some computational problem, that incrementally builds candidates to the solutions, and abandons each partial candidate *c* ("backtracks") as soon as it determines that *c* cannot possibly be completed to a valid solution.

Recursion and Factorial

Consider the **Factorial Function:** Product of all the integers between 1 to n.
n! = 1 if n= 0
n! = n* (n-1) * (n-2) * . . . * 1 if n > 0

To avoid the shorthand definition for n! We have to list a formula for n! for each values of n separately.
0! = 1
1! = 1
2! = 2 * 1
3! = 3* 2* 1
. . . .

To avoid the shorthand and to avoid the infinite set of definitions, yet to define function precisely, we can write an algorithm that accepts an integer n and returns the value of n!

```
prod = 1;
   for(x = n; x > 0; x--)
      prod * = x;
return (prod);
```

This process is **Iterative.**
Iterative: explicit repetition of some process until a certain condition is met.

Let us take a closer look:
0! = 1
1! = 1 * 0!
2! = 2 * 1!
3! = 3 * 2!
4! = 4 * 3!
…
…
And so on…

Using mathematical notation, we can write it as:
n! = 1 if n =0
n! = n * (n-1)! If n > 0

This defines factorial in terms of itself.

Looks like a circular definition. A definition which defines an object in terms of a simpler case of itself is called a **recursive definition.**

Usage:

```
1     5! = 5 *4!
2          4! = 4 * 3!
3               3! = 3 * 2!
4                    2! = 2 * 1!
5                         1! = 1 * 0!
6                              0! = 1
```

Now start back tracking

```
6'                             0! = 1
5'                         1! = 1 * 1
4'                   2! = 2 * 1
3'              3! = 3 * 2
2'         4! = 4 * 6
1'    5! = 5 * 24
```

Which finally results in 120

Algorithm:

```
if(n == 0)
   fact =1;
else {
    x = n – 1
   find the value of x! call it y;          re-executing the algorithm with value x
   fact = n * y;
}
```

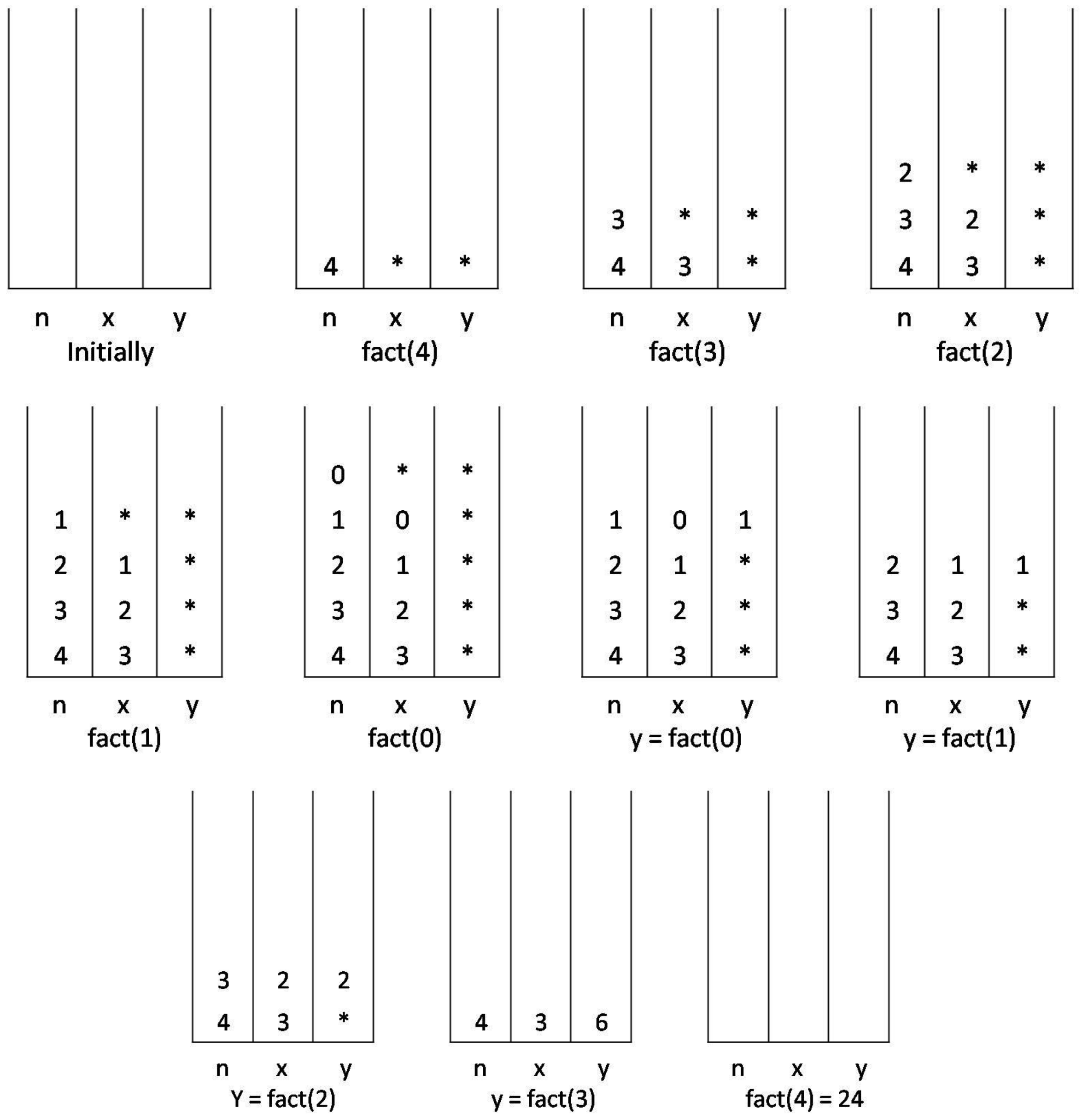
Note: This example is to introduce recursion, not as more effective method of solving the particular problem.

Implementing in C language:

```c
int fact (int n)
{
   int x, y;
   if(n ==0)
      return 1;
   x = n – 1;
   y = fact(x);
   return (n * y);
}
```

Recursive calls maintains stack to keep the local copies of variable of each call which is invisible to the user. On each call a new allocation of variable is pushed onto the stack. When the function returns, the stack is popped.

Below diagram summarizes the process for the call : fact(4)



Recursive Version of Pingala Series Numbers:

```
int pingala(int n)
{
   int x, y;
   if( n <= 1)
      return n;
   x = pingala (n-1);
   y = pingala (n-2);
   return (x + y);
}
```

Writing recursive programs:

- Find the trivial case
- Find a method of solving a complex case in terms of a simpler case
- The transformation of complex to simpler case should eventually result in trivial case

However it is not easy to write recursive codes. It only comes by lot of practice and well understanding of the procedure. One should also be able to analyse the difference between Iterative and recursive procedure.

Example: Count the number of digits in binary representation of a decimal

```
ALGORITHM Binary(n)
// Counts the number of digits in binary representation of a given positive decimal integer
// Input: a positive decimal integer
// Output: Number of digits in a binary representation of a given positive decimal integer
if n=1
   return 1
else
  return Binary (n/2) + 1
```

[**Note:**

The iterative version of above algorithm would be written as:

```
count ← 1
while n > 1
   count ← count + 1
   n ← n / 2
return count ]
```

The Towers of Hanoi Problem

Task:

There are 3 pegs and N disks. The larger disk is always at the bottom. We need to move the N disks from A to C using B as auxiliary with the constraint that the smaller disk is always at the top.

Solution:

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

(If you are not able to understand above procedure, apply the solution on 3 disks problem. The procedure will follow.)

Designing a program:

- Design a better presentation of the solution
- Think on what will be the input and output to the program
- Present the output in user understandable way
- The proper input / output format may make the program design much simpler

Program:

```
#include <stdio.h>
#include <stdlib.h>

void towers(int, char, char, char);

int main()
{
   int n;
   printf("Enter the number of Disks to be moved\n");
   scanf("%d", &n);
   towers(n, 'A', 'C', 'B');
   return 0;
}

void towers(int n, char from, char to, char aux)
{
   if( n == 1)
   {
      printf("Move disk 1 from %c to %c\n",from, to);
      return;
   }
   // Move top n-1 disks from A to B using C as auxiliary
   towers(n-1, from, aux, to);

   // Move remaining disk from A to C
   printf("Move disk %d from %c to %c\n", n, from, to);

   // Move n-1 disks from B to C using A as auxiliary
   towers(n-1, aux, to, from);
}
```

Efficiency of Recursion

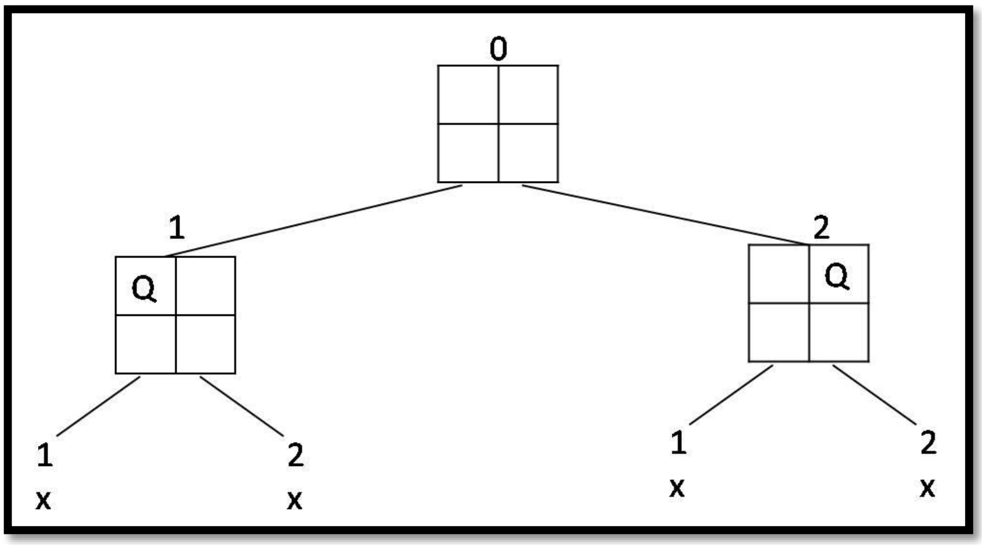
- Non recursive version of a program will execute more efficiently in terms of time and space
- The stacking activity hinders the performance of recursive procedures
- Some of the variables can be identified which do not have to be moved into stack, and recursion can be simulated by pushing and popping only the relevant variables
- Sometimes recursion is the most natural and logical way of solving the problem – towers of Hanoi
- At some cases, it is better to create a non-recursive version by simulating and transforming the recursive version than attempting to create a non-recursive solution from the problem
- Some of the function calls can be replaced with in line codes so as the reduce the usage of stack

Backtracking Example: N Queens Problem

Place 'n' queens on an 'n x n' board such that no queen attacks immediate diagonally, vertically or horizontally.
For a 1 queen problem, we need to place the queen on 1 x 1 board. And the solution is:

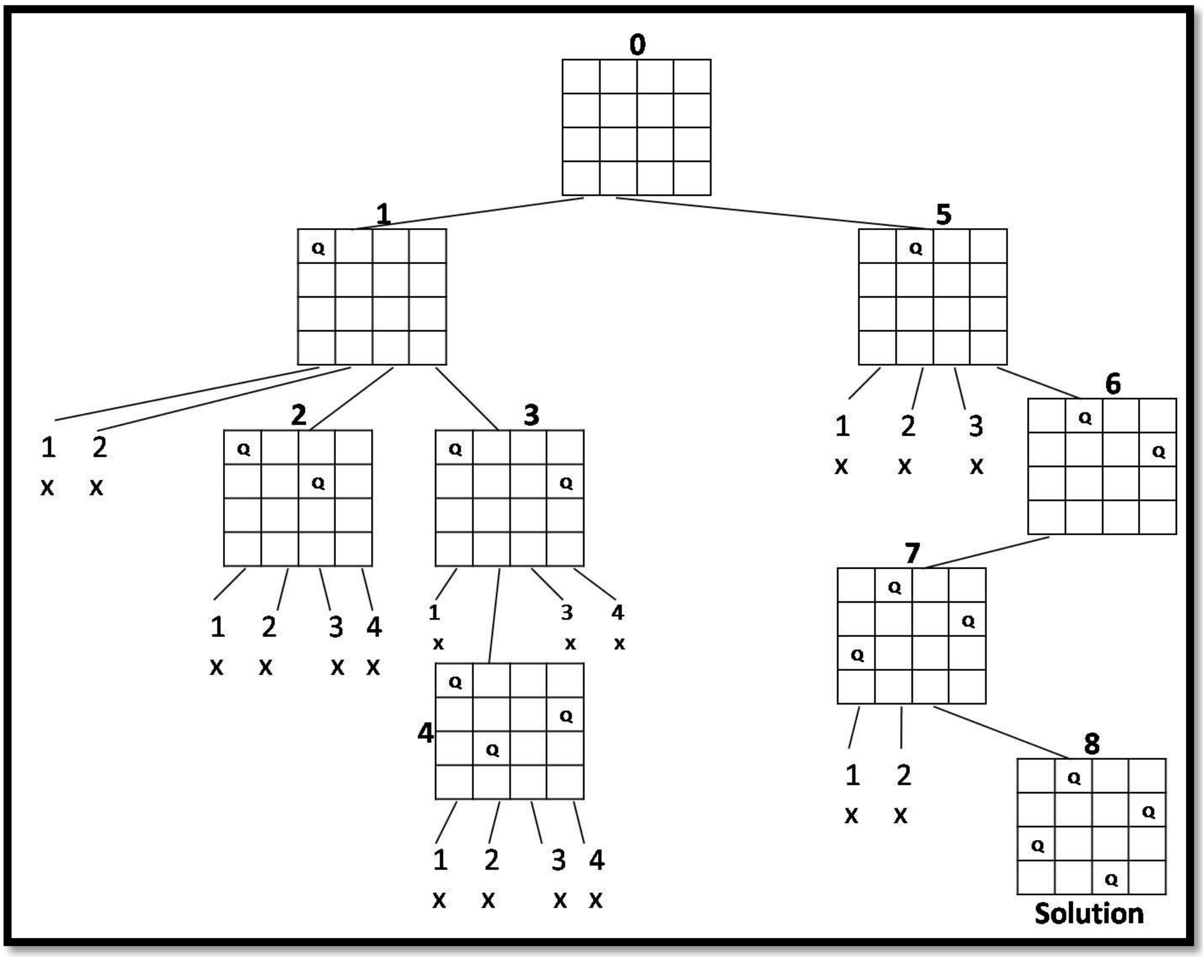
Q

For a 2 queen and 3 queen problem, we do not have solution. Let us look at the procedure for 2 queen problem.



Above shown is the 'state space tree' for 2 queen problem. Nodes in the tree are solution instances in the order generated. '0' is the initial node in state space tree. Node '1' has placed successfully the first queen. 'x' indicates an unsuccessful attempt to place a queen in the indicated column. As we cannot place the 2nd queen in node '1', we backtrack and place 1st queen in next avail position. However that fails too!

State space tree for 4 Queen Problem:



Note:
Refer class notes for more recursion examples / subset sum problem on backtracking.

For notes on structures, download ebook:
https://www.smashwords.com/books/view/644937

Design of a Programmer:
https://www.smashwords.com/books/view/639609

~*~*~*~*~*~*~*~*~