Data Structures and Algorithms

School of CSE, KLE Tech. University
Prakash Hegade
2017-18

CONTENTS

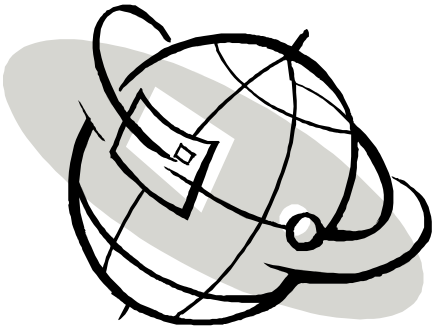
"Learn the basics listen ten,
You will know, why and when,
Switch your mode, get to code,
That's the way you build your road."

-PH

1. Introduction to Data Structures and Recursion (Contd..,)

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 behavior 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

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

Let us boil down to understanding recursion using the factorial problem.

Factorial of 'n' is: Product of all the integers between 1 to n.

n! = 1 if n= 0

n! = n* (n-1) * (n-2) * . . . * 1 if n > 0

Above is a short hand definition for factorial.

We can also go for a formula definition. We have to list down the formula for n!, for each values of 'n' separately.

0! = 1
1! = 1
2! = 2 * 1
3! = 3* 2* 1
….

We can avoid the shorthand and the infinite set of definitions using formula and go for something more precise using an algorithm.

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 at the factorial problem. We can also define factorial as:
0! = 1
1! = 1 * 0!
2! = 2 * 1!
3! = 3 * 2!
4! = 4 * 3!
…
And so on…

Using mathematical notation, we can write above process as:

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

As we observe above, 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

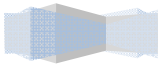
Writing Algorithm for above:

```
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;
}
```

Implementing in C language:

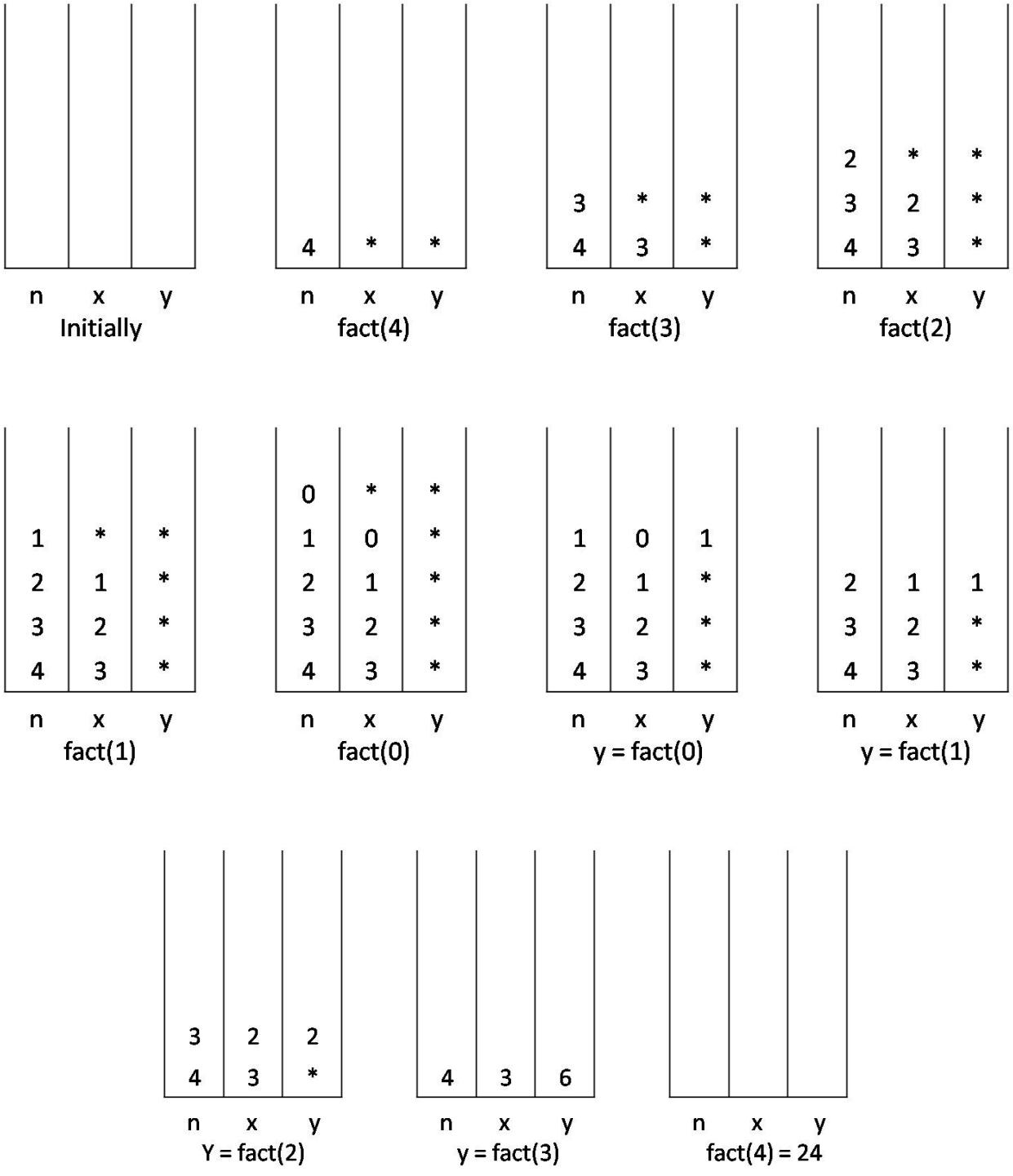
```
int fact (int n)
{
int x, y;
if(n ==0)
return 1;
   x = n – 1;
   y = fact(x);
return (n * y);
}
```

Kindly note that: this example is to introduce recursion and not as effective method of solving a particular problem.



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 copy 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)



Recursion Examples

1. Binary Search

```
int binsrch(int a[], int x , int low, int high)
{
   int mid;
   if(low > high)
      return -1;
   mid = (low + high ) / 2;
   return ( x == a[mid] ? mid : x < a[mid] ? binsrch(a, x, low, mid -1) : binsrch(a, x, mid+ 1, high));
}
```

2. Pingala Series

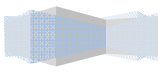
```
int pingala(int n) {
   int x, y;
   if( n <= 1)
      return n;
   x = pingala (n-1);
   y = pingala (n-2);
   return (x + y);
}
```

3. Sum of N Natural Numbers

```
int sum(int num)
{
   if (num!=0)
      return num + sum(num-1);
   else
      return num;
}
```

4. Calculate Length of a String

```
int string-length(char *str)
{
   static int length=0;
   if(*str!=NULL) {
      length++;
      string-length(++str);
   }
   else
      return length;
}
```



5. Story Illustration

A child couldn't sleep, so her mother told a story about a little frog,
who couldn't sleep, so the frog's mother told a story about a little bear,
who couldn't sleep, so bear's mother told a story about a little weasel
...who fell asleep.
...and the little bear fell asleep;
...and the little frog fell asleep;
...and the child fell asleep.

6. Test if Palindrome

```
bool is-palindrome(char* s, int len)
{
 if(len < 2)
   return TRUE;
 else
   return s[0] == s[len-1] && is-palindrome(&s[1], len-2);
}
```

7. Count the Number of Digits in a given number (Example: 56234 has 5 digits)

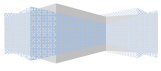
```
int count-digits(int num)
{
   static int count=0;
   if(num>0) {
      count++;
      count-digits(num/10);
   }
   else
      return count;
}
```

8. Count the Number of Digits in Binary representation of a given positive decimal Integer
[example: 4 is represented in binary using three digits]

```
int binary(int n)
{
   if (n == 1)
      return 1;
   else
      return binary(n/2) + 1;
}
```

9. Man or Boy Test

- By Donald Knuth
- Was used to evaluate ALGOL 60 programming language



- Tests identified compilers that correctly implemented recursion and non local references and those did not

10. Sum of Array Elements

```
int array-sum(int a[ ], int start, int stop)
{
  if (start == stop)
    return a[stop];
  else
    return (a[stop] + array-sum(a, start+1, stop);
}
```

Note: Look out class notes for tracing

11.

In order to **Understand Recursion**
One must first Understand **Recursion**

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

Recursive Chains

A recursive function need not call itself directly. Rather, it may call itself indirectly.

```
function-a ( formal parameters) {
  .
  .
  function-b(arguments);
}
```

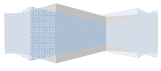
```
function-b(formal parameters) {
  .
  .
  function-a(arguments);
}
```

More than 2 routines may participate in a recursive chain.

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

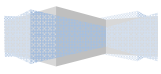
(If you are not able to understand above procedure, apply the solution on 3 disks problem. The procedure will follow.)

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 Examples

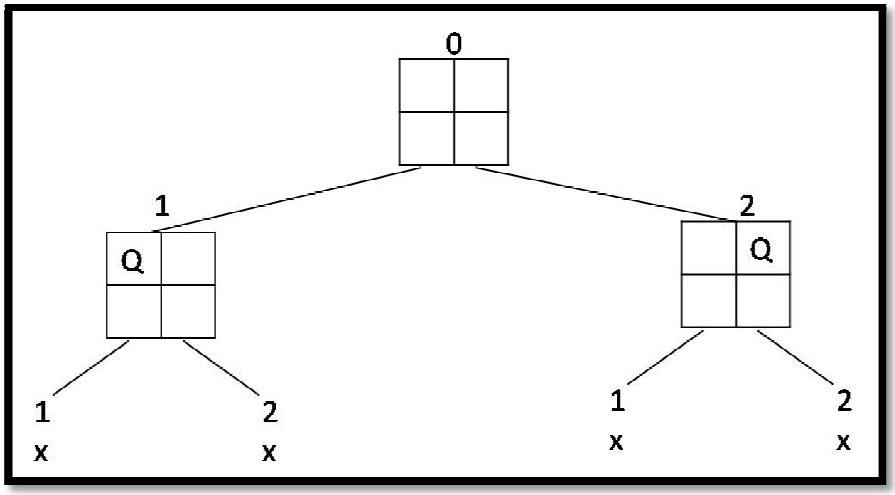
1. N Queens Problem

Place 'n' queens on a '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:

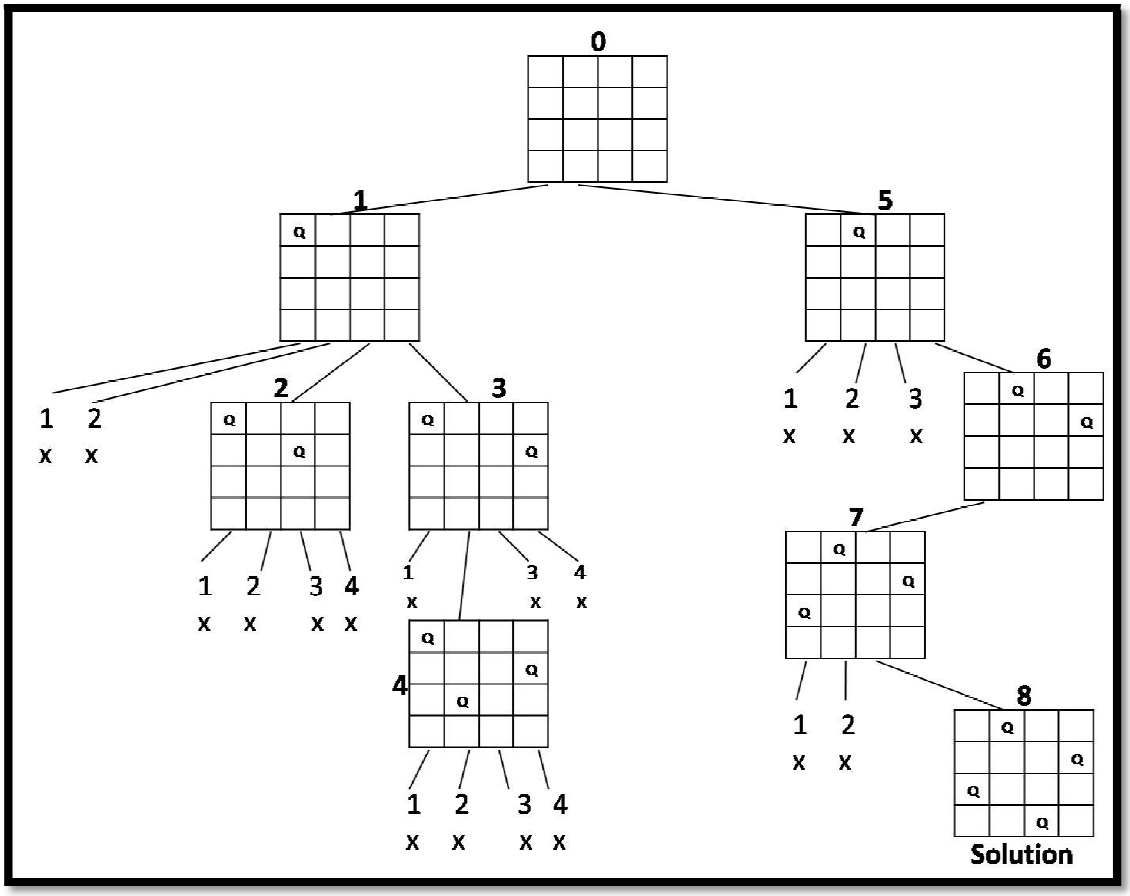
Q

For a 2 queen and 3 queen problem, we do not have solution.
Let us look at the state space tree of 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:



2. Subset Sum

Problem: Find subset of elements that are selected from a given set whose sum adds up to a given number K.

Assumptions:

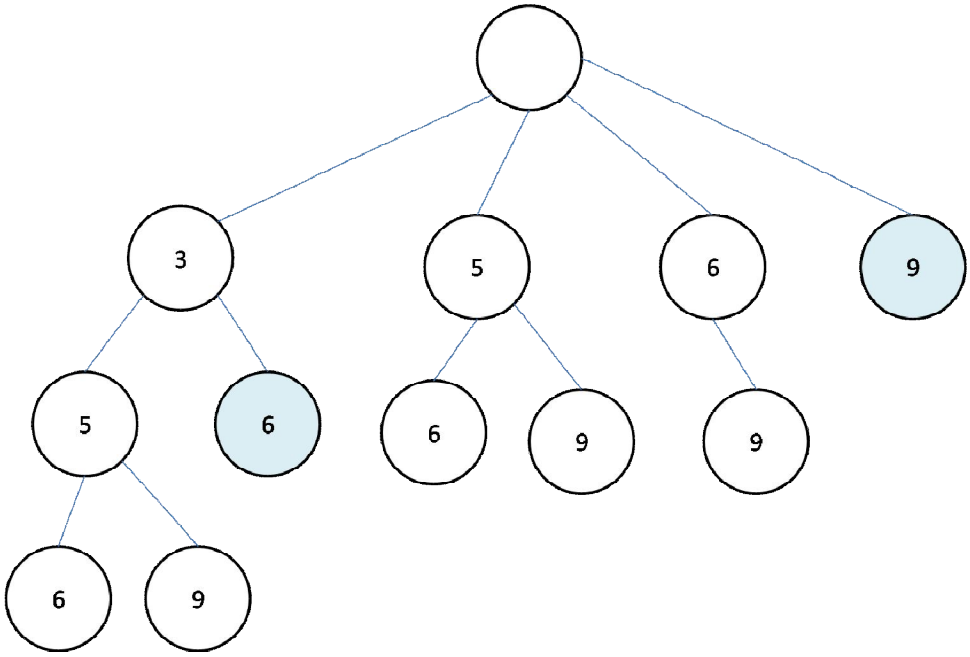
- Set has all non negative values
- Input set is unique

We can use an exhaustive search technique by generating the power set of given set, adding all the subsets and then matching with number K. However the process expensive as compared to backtracking.

Using exhaustive search we generate all subsets irrespective of whether they satisfy given constraint or not. Backtracking can be used to make a systematic consideration of the elements to be selected.

Example:
A = {3, 5, 6, 9}
K = 9

A subset sum tree can be seen below. The sum is calculated from the top most node of the tree (root) till the leaf node. The leaf nodes with sum = 9 are shaded. The tree halts generating further nodes when sum exceeds 9.



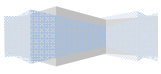
Refer class notes for more examples and details.

Additional Reading Materials

Function Pointer

Pointer can not only be an address of another variable, it can also be extended to hold a function address.

What do we mean by function address?
Every function definition that is written while executing gets loaded into memory. This function is always recognized and called by the name given to it. There is also an alternate mechanism to call the function. The way we have value variable and address variable, a function can also be called by its name (the traditional way) or address where it is loaded.



We use a function pointer to hold the starting address of the function definition. This address is entry point to the function. Once the pointer points to this function, the function can be invoked using this pointer.

Function Pointer Syntax:
return_type (*fp) (parameters_type);

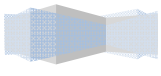
Using function pointers we can pass name of the function as parameter to a function.

Program: String compare program to demonstrate the usage of function pointers
(Go with the numbering (Read 01, Read 02 and Read 03) and understand the program)

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

int strcmp(char *ptr1, char *ptr2 )
{
  while(*ptr1 == *ptr2)
  {
    if(*ptr1 == '\0')
      break;
    ptr1++; ptr2++;
  }
  return (*ptr1 - *ptr2);
}
```

// Read 02: Look at how the first parameter is caught.
/*
A parameter is always caught based on the type. As the first parameter is function, we catch it the way function is defined. Look at syntax of function pointer.
We first need to specify the return type followed by is name of the function. As name of the function is an address, we need to use a *. Here '*cmp' is a pointer which will hold the starting address of the function strcmp. Then we write the parameters.
Let's see the mapping:
int strcmp(char *ptr1, char *ptr2) mapped to
int (*cmp) (char *, char *)
/*

```
void compare_and_display(int (*cmp) (char *, char *), char *a, char *b)
{
  int diff = 0;

  // Read 03: Now the function strcmp can be called by using cmp. Can you say why?
  diff = cmp(a,b);
```



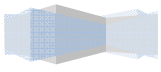
```
   if(diff == 0)
      printf("Two strings are equal\n");
   else if(diff > 0)
      printf("First string is greater than second string\n");
   else
      printf("Second string is greater than first string\n");
}

int main()
{
   char str1[20], str2[20];
   printf("Enter the two strings to be compared\n");
   scanf("%s  %s", str1, str2);

   // Read 01: Look at the first parameter to the function call. It is name of the function.
   compare_and_display(strcmp, str1, str2);
   return 0;
}
```

Unions

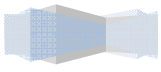
Below given is a program, followed by properties. First go through the program, and then read the properties of Unions. Revisit the program again and understand the unions better. Do not forget to run the code and understand.

A program to demonstrate the Unions

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

enum type
{
   INT, FLOAT, STRING
} tag;

union sym_value
{
  int ival;
  float fval;
  char *sval;
};
```



```
int get_input(union sym_value *u)
{
   int choice = 0;
   printf("Enter\n1- initilaize int\n2-initilaize float\n3-initilaize string\n");
   scanf("%d", &choice);
   switch(choice)
   {
      case 1: u->ival = 20;
            tag = INT;
            break;

      case 2: u->fval = 200.0;
            tag = FLOAT;
            break;

      case 3: u->sval = "string";
            tag = STRING;
            break;
   }
   return tag;
}

int main()
{
   union sym_value u1;
   union sym_value u2;
   int result = 0;

   printf("Size of u1 Variable is... %d\n", sizeof(u1));
   u1.ival = 10;
   u1.fval = 10.0;
   u1.sval = "abc";
   printf("Integer is..%d\n", u1.ival);
   printf("Float is..%f\n", u1.fval);
   printf("String is..%s\n", u1.sval);

   u2 = u1;
   printf("String of u2 variable is..%s\n", u2.sval);

   result =  get_input(&u2);
   printf("The data initialized in Function is..\n");
```



```
  if(result == 0)
     printf("Integer: %d\n", u2.ival);
  else if(result == 1)
     printf("Float : %f\n", u2.fval);
  else
    printf("String: %s\n", u2.sval);
  return 0;
}
```

Characteristics of UNIONS:

- The size of memory allocated by the compiler is the size of the member that occupies largest space
- Memory allocated is shared by individual members of union
- The address is same for all the members of a union
- Only one member can be accessed at a time
- Only the first member of a union can be initialized

Bit Fields

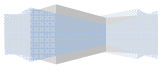
A group of several bits can be packed together using a structure. A member of a structure that is composed only of a specified number of bits is called bit-fields.

Syntax:

```
struct tag_name
{
        data_type  member01: bit_length;
        data_type  member02: bit_length;
        …
        …
};
```

Program to demonstrate the Bit Fields

```
#include <stdio.h>
struct student
{
  char *name;
  unsigned sem : 3;
  unsigned courses_registered : 3;
  unsigned eligibility : 1;
};

int main()
{
```



```
    struct student s1;
    int sem, courses_registered, eligibility;

    s1.name = "aaa";
    printf("Enter the sem, courses registered and is student eligible data\n");
    scanf("%d%d%d",&sem,&courses_registered, &eligibility);

    s1.sem = sem;
    s1.courses_registered = courses_registered;
    s1.eligibility = eligibility;

    printf("The entered details are...\n");
    printf("Name: %s\nSem: %d\nCourses Registered: %d\nEligibility: %d\n", s1.name, s1.sem,
s1.courses_registered, s1.eligibility);

    return 0;
}
```

Characteristics:

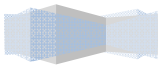
- Addresses of bit fields cannot be accessed. So, we cannot read the values into the bit fields using scanf()
- Pointers cannot be used to access bit fields
- Bit fields cannot be declared as static
- Bit fields should be assigned the values within the range. Otherwise the behavior of the program will be unpredictable
- No field can be longer than 1 long word (32 bits)
- It is not possible to declare array of bit fields

Proof by Induction

A proof by induction is just like an ordinary proof in which every step must be justified.

However it employs a neat trick which allows you to prove a statement about an arbitrary number n by first proving it is true when n is 1 and then assuming it is true for n=k and showing it is true for n=k+1.

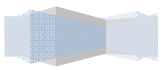
The idea is that if you want to show that someone can climb to the nth floor of a fire escape, you need only show that you can climb the ladder up to the fire escape (n=1) and then show that you know how to climb the stairs from any level of the fire escape (n=k) to the next level (n=k+1).



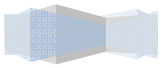
Summary program

A Program summarizing most of the concepts covered in chapter 01.

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

union charcteristics
{
   int padaku;
   int cool;
   int CSEian;
};

struct nature
{
   int tag;
   union charcteristics c;
};

struct student
{
   char name[20];
   char *dept;
   int roll;
   unsigned prep_status : 1;
   struct nature n;
};

int main()
{
   struct student s;
   strcpy(s.name, "Vishwa");
   s.dept = "CSE";

   printf("Enter Roll Number\n");
   scanf("%d", &s.roll);

   s.prep_status = 0;

   printf("Enter Student behavior\n");
   printf("1-Padaku\n2-Cool\n3-CSEian\n");
```



```
scanf("%d", &s.n.tag);

if(s.n.tag == 1)
   s.n.c.padaku = 1;
else if(s.n.tag == 2)
   s.n.c.cool = 1;
else
   s.n.c.CSEian = 1;

printf("Student name is..%s \n", s.name);
printf("Student department is..%s \n", s.dept);
printf("Student roll is..%d \n", s.roll);
printf("Student preparedness is..%d \n", s.prep_status);
if(s.n.tag == 1)
   printf("Student is padaku\n");
else if(s.n.tag == 2)
   printf("Student is cool\n");
else
   printf("Student is CSEian\n");


printf("Enter the top 05 coders of the week roll numbers\n");
int array[5];
int index;
int *p;
for(index =0; index<5;index++) {
   scanf("%d", (array+index));
}

p = array;
printf("The Top coders of the week are..\n");
for(index =0; index<5;index++) {
   printf("%d\t", *p);
   p++;
}

printf("\n\nThe sleepy student of the class is..\n");
char sleepy_student[30] = "ZZZ";
char *c;

c = sleepy_student;
```



```
 while(*c != '\0')
   {
      printf("%c", *c);
      c++;
   }

   printf("\nThese benches are always occupied by same students\n");
   int *arr[5];
          int st1 = 1, st2 = 10, st3 = 18, st4 = 7, st5 = 21;
          int index1;
          arr[0] = &st1;
          arr[1] = &st2;
          arr[2] = &st3;
          arr[3] = &st4;
          arr[4] = &st5;

          for(index1 = 0; index1 < 5; index1++)
                printf("%d\n", *(arr[index1]));

   return 0;
}
```

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