Data Structures and Algorithms (CS-2001)

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY

School of Computer Engineering

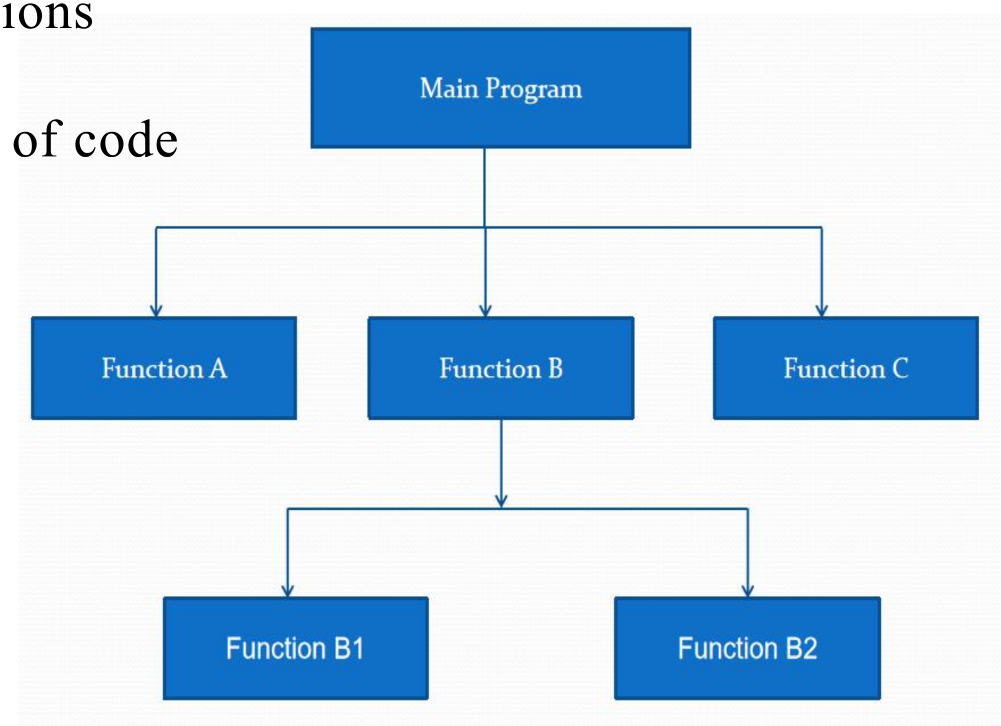
Strictly for internal circulation (within KIIT) and reference only. Not for outside circulation without permission

4 Credit

Lecture Note

Function

- The program becomes large and complex
- A lengthy program can be coded separately into sub-programs called as Functions
- For repetition of certain part of code



Function Definition

The general form of function definition statement in C is as follows:

```
return_data_type function_name(data_type variable1,...) Function header
{
    declarations;
    statements;                Function Body
    return(expression);
}
```

- **function_name** :
  - This is the name given to the function
  - it follows the same naming rules as that for any valid variable in C.
- **return_data_type**:
  - This specifies the type of data given back to the calling construct by the function after it executes its specific task.

Passing Values between Function

```
int main()
{
        int a, b, c, sum ;

        printf ( "\nEnter any three numbers " ) ;
        scanf ( "%d %d %d", &a, &b, &c ) ;

        sum = calsum ( a, b, c ) ;

        printf ( "\nSum = %d", sum ) ;

        return 0;
}
```

```
calsum ( int x, int y, int z )
{
        int d ;
        d = x + y + z ;
        return  d  ;
}
```

Passing Values between Function

```
int main()
{
        int a, b, c, sum ;

        printf ( "\nEnter any three numbers " ) ;
        scanf ( "%d %d %d", &a, &b, &c ) ;

        sum = calsum ( a, b, c ) ;

        printf ( "\nSum = %d", sum ) ;

        return 0;
}
```

```
int calsum ( int x, int y, int z )
{
        int d ;
        d = x + y + z ;
        return d ;
}
```

- a, b, c are called **actual** arguments
- x, y and z are called **formal** arguments
- **main** is the **calling** function and **calsum** is the **called** function

Function return statement

Function return statement

There is **no restriction** on the number of return statements that may be present in a function. Also, the return statement need not always be present at the end of the called function. The following program illustrates these facts.

```
int fun( )
{
      char ch ;

      printf ( "\n Enter any alphabet " ) ;
      scanf ( "%c", &ch ) ;

      if ( ch >= 65 && ch <= 90 )
           return ( ch ) ;
      else
           return ( ch + 32 ) ;
}
```

Scope Rule of Functions

```
display ( int j )
{
      int k = 35 ;
      printf ( "\n%d", j ) ;
      printf ( "\n%d", k ) ;
}

 int main( )
 {
      int i = 20  ;
      display ( i ) ;
      return 0;
 }
```

In this program is it necessary to pass the value of the variable i to the function display( )? Will it not become automatically available to the function display( )? **No**.Because by default the scope of a variable is local to the function in which it is defined. The presence of **i** is known only to the function main( ) and not to any other function. Similarly, the variable k is local to the function display( ) and hence it is not available to main( ). That is why to make the value of i available to display( ) we have to explicitly pass it to display( ). Likewise, if we want k to be available to main( ) we will have to return it to main( ) using the return statement. In general we can say that the scope of a variable is local to the function in which it is defined.

Function Call by Value and Call by Reference

Whenever we called a function and passed something to it we have always passed the ‘values’ of variables to the called function. Such function calls are called ‘**calls by value**’. By this what we mean is, on calling a function we are passing values of variables to it. The examples of call by value are shown below:

```
sum = calsum ( a, b, c ) ;
```

We have also learnt that variables are stored somewhere in memory. So instead of passing the value of a variable, can we not pass the location number (also called address) of the variable to a function? If we were able to do so it would become a ‘**call by reference**’.

Arguments can generally be passed to functions in one of the two ways:

- sending the values of the arguments
- sending the addresses of the arguments

Function Call by Value and Call by Reference

Call by Value

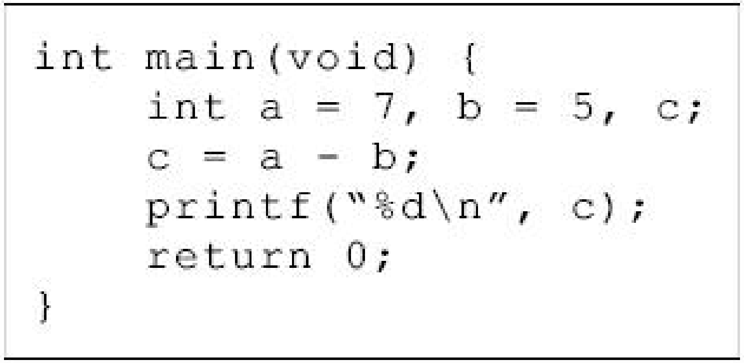
```
swapByValue ( int x, int y )
{
        int t ;
        t = x ;
        x = y ;
        y = t ;
        printf ( "\nx = %d y = %d", x, y ) ;
}

int main( )
{
        int a = 10, b = 20 ;
        swapByValue( a, b ) ;
        printf ( "\na = %d b = %d", a, b ) ;
        return 0;
}
```

Call by Reference

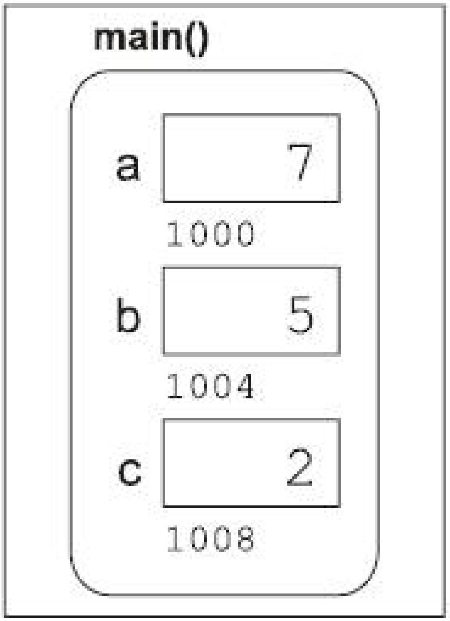
```
swapByRef ( int *x, int *y )
{
        int t ;
        t = *x;
        *x = *y ;
        *y = t ;
        printf ( "\nx = %d y = %d", *x, *y ) ;
}

int main( )
{
        int a = 10, b = 20 ;
        swapByRef( &a, &b ) ;
        printf ( "\na = %d b = %d", a, b ) ;
        return 0;
}
```

Pointers

Storing Variable in Memory



```
int main(void) {
    int a = 7, b = 5, c;
    c = a - b;
    printf("%d\n", c);
    return 0;
}
```



The value inside a box indicates the value in that memory location.

The value under a box is the memory address of that box.

Pointers

- When we declare a variable and initialize it,
  - int quantity = 179 ;

quantity	variable
179	value
5000	address

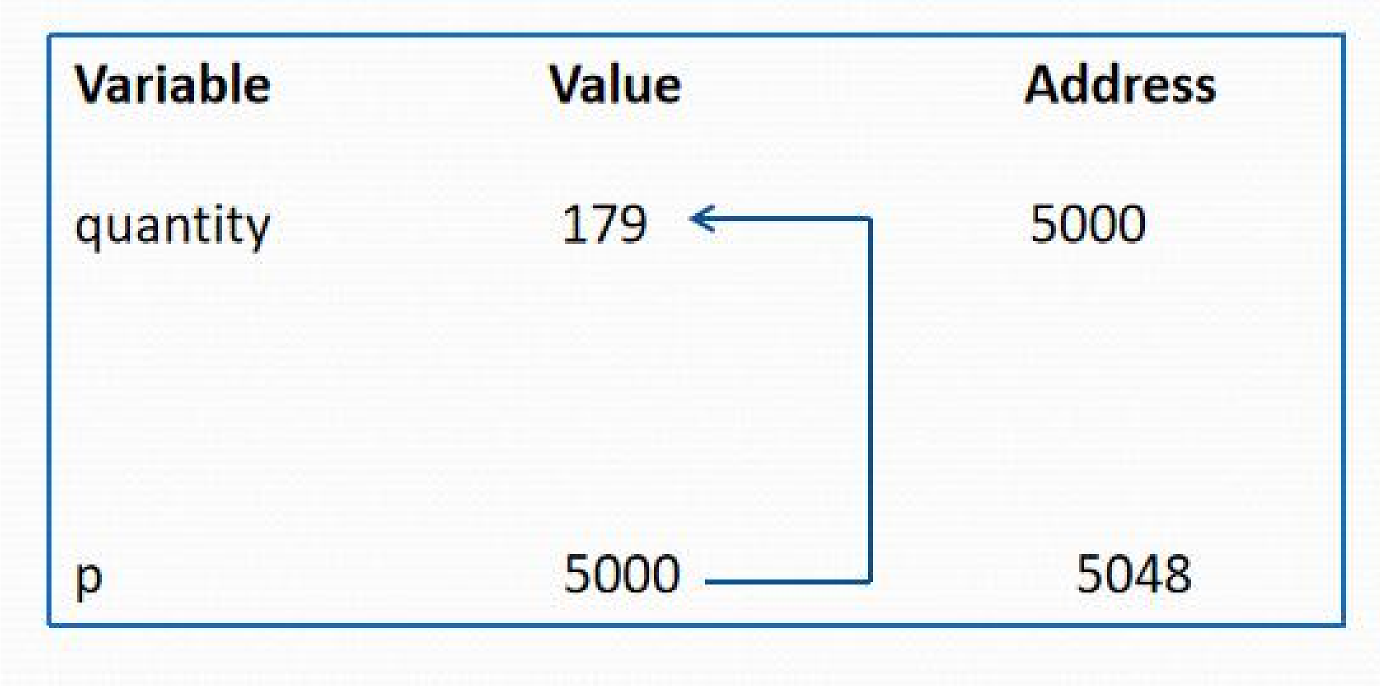
Since memory address are simple numbers they can be assigned to some other variables

Pointers

Pointers

- Pointer is an entity which contains memory address
- Pointers can be used to access data stored in memory
- There are two operators for pointers
  - address operator ‘&’
  - Dereference operator ‘*’ (value of operator)

Pointers



Variable	Value	Address
quantity	179	5000
p	5000	5048

Accessing Address of a Variable

```
main( )
 {
  int i=3;
  printf("\n Address of i = %u",&i);
  printf("\n Value of i = %d",i);
}
```

OUTPUT :

```
Address of i = 6485
Value of i =3
```

Declaring pointer Variables

- The declaration is done as follows

  *data_type *pt_name;*

  - * tells that pt_name is a pointer variable
  - pt_name needs memory location
  - pt_name points to a variable of type *data_type*

- Remember data_type tells the type of data pointer will point to and not the value of pt_name

- Example
  - int *p;
  - float *x;

Initialization of pointer variables

- int quantity;
  ```
  int *p; //declaration
  p=&quantity; //initialization
  ```

- int x, *p=&x; // all steps in one

- // Assignment of NULL or 0 (zero)
  ```
  int *p=NULL;
  int *p=0;
  ```

- // this is invalid, absolute address
  ```
  int *p=5360;
  ```

Accessing Variable Through its Pointer

- * operator known as
  - Indirection operator
  - Dereference operator

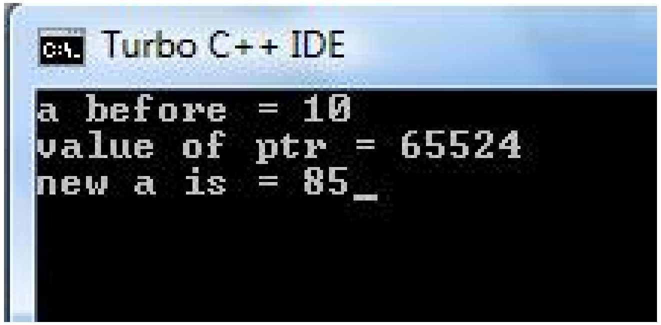
- Example

```
int a,*p, n;    //declaration
a=179;          // variable assignment
p=&a;           // pointer initialization
n=*p;           //value at address operator
```

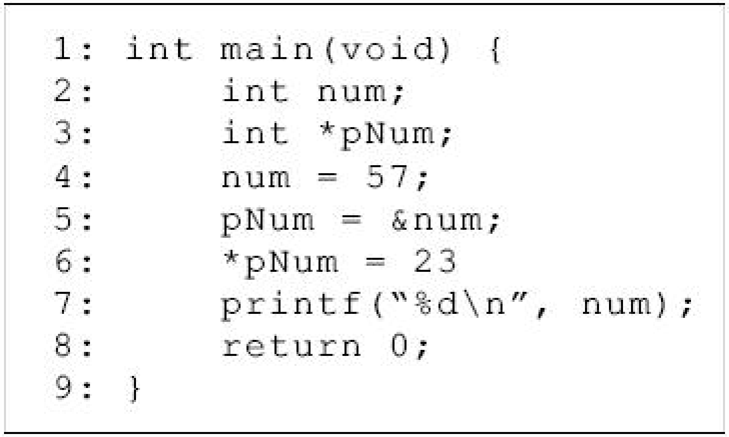
  - When * is placed at declaration it is used creating a pointer
  - * placed on right hand side of expression it returns the value at a address. In this case it will return 179

Accessing Variable Through its Pointer

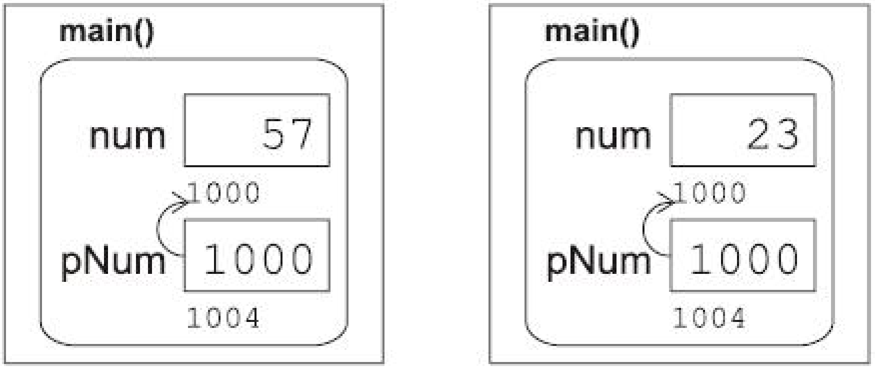
```
#include<stdio.h>
main()
{
int a=10;
int *ptr;
printf("a before = %d",a);

ptr=&a;
printf("\nvalue of ptr = %u", ptr);

*ptr=85;
printf("\nnew a is = %d",a);
getch();
}
```



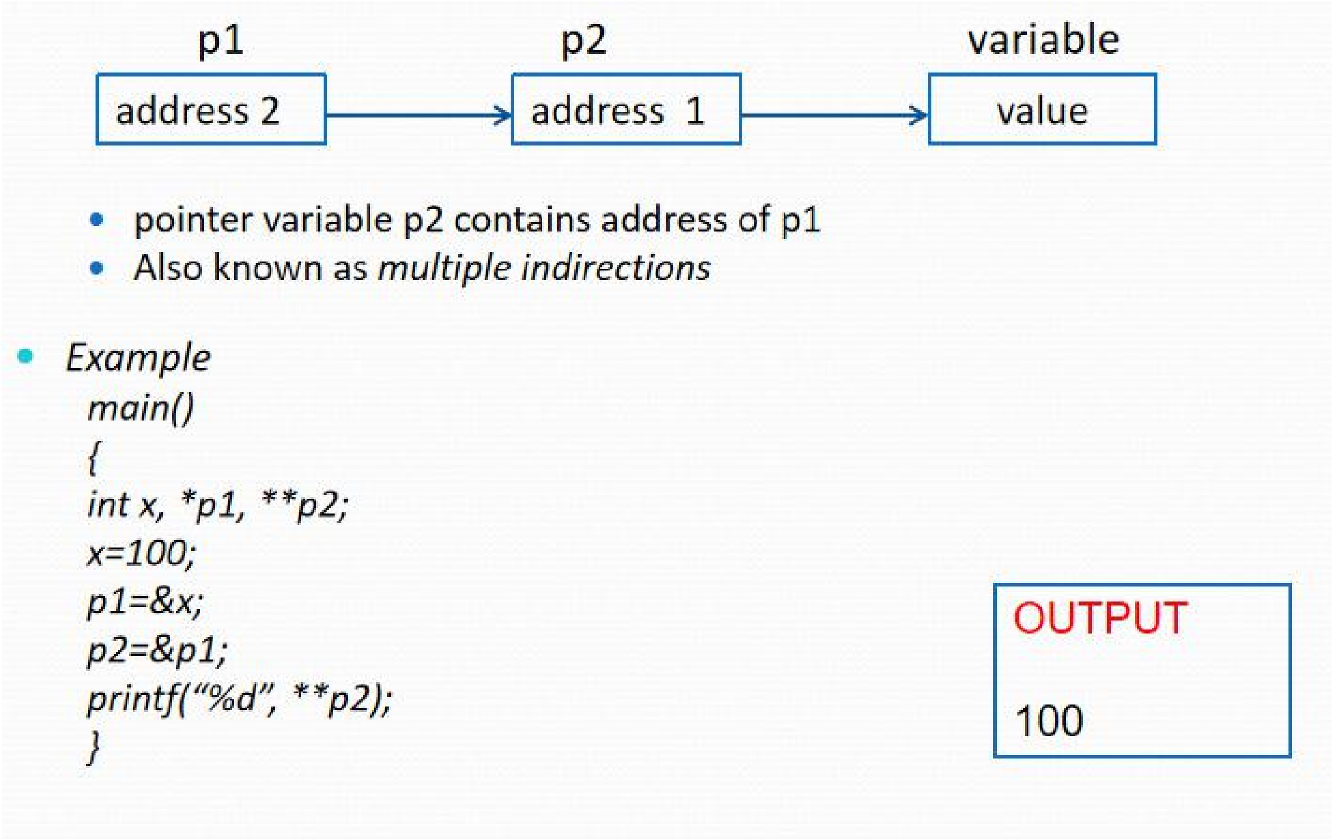
```
a before = 10
value of ptr = 65524
new a is = 85_
```



```
int main(void) {
      int num;
      int *pNum;
      num = 57;
      pNum = &num;
      *pNum = 23
      printf(“%d\n”, num);
      return 0;
}
```



Chain of Pointers



p1 p2 variable

address 2 → address 1 → value

- pointer variable p2 contains address of p1
- Also known as *multiple indirections*

- *Example*

```
main()
{
int x, *p1, **p2;
x=100;
p1=&x;
p2=&p1;
printf("%d", **p2);
}
```

OUTPUT

```
100
```

Pointer Expression

- Following statements are valid

```
y=*p1 * *p2;
sum=sum+*p1;
z=5*(-(*p2))/ *p1;
```

- Two pointers cannot be used for division or multiplication

Pointer Increments and Scale Factors

- *Pointers can be incremented like*
  - p1=p1+2;
  - p1=p1+1;

- *Expression like*
  - p1++
  - will cause pointer to next value of its type

Structure

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in Cthat allows to combine data items of **different kinds**.

Structures are used to **represent** a **record**. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

- Title
- Author
- Subject
- Book ID

Defining the Structure:

```
struct [structure tag]
{

  member definition;
  member definition;
  ...
  member definition;
} [one or more structure variables];
```

Book Structure

```
struct Books
{
  char title[50];
  char author[50];
  char subject[100];
  int book_id;
} book;
```

Access Structure Elements

```
/* Declare Book1 of type Book */
struct Books Book1;

/* Declare Book2 of type Book */
struct Books Book2;
Book1.title = "DSA";
Book2.book_id = 6495700;
```

Union

Unions are quite similar to the structures in C.Union is also a **derived type** as structure. Union can be defined in same manner as structures just the keyword used in defining union in **union** where keyword used in defining structure was **struct**.

You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining the Union:

```
union [union tag]
{

  member definition;
  member definition;
  ...
  member definition;
} [one or more union variables];
```

Data Union

```
union Data
{
  int i;
  float f;
  char str[20];
} data;
```

Access Union Elements

```
/* Declare Book1 of type Book */
struct Data data;

data.i = 10;
data.f = 34.72;
data.str = "C Programming"
```

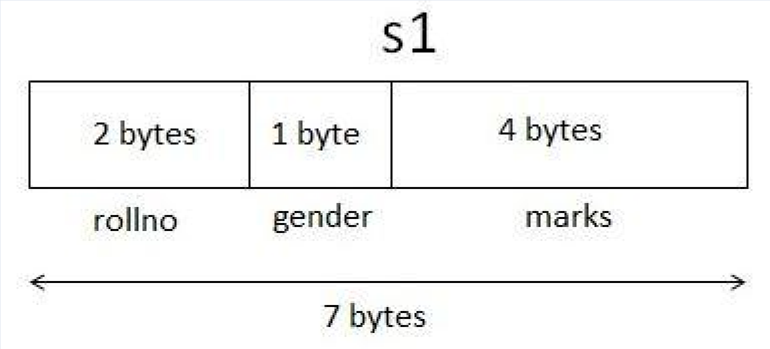
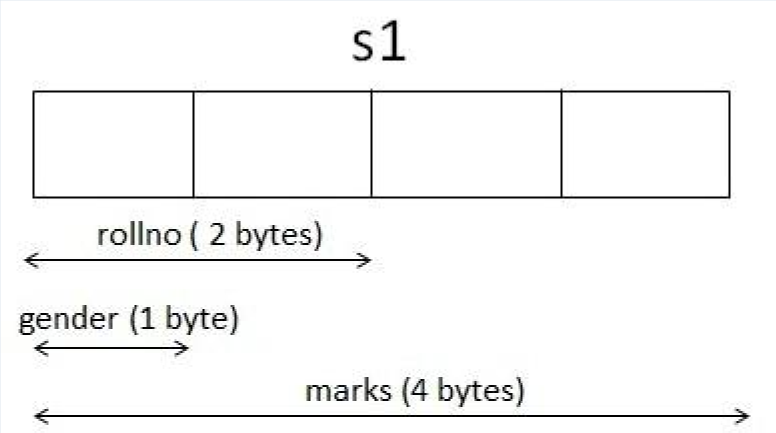
Difference b/w Structure & Union

Structure	Union

In structure each member get separate space in memory. Take below example.
 	
struct student
{
int rollno;
char gender;
float marks;
} s1;
 	
The total memory required to store a structure variable is equal to the sum of size of all the members. In above case 7 bytes (2+1+4) will be required to store structure variable s1.
	In union, the total memory space allocated is equal to the member with largest size. All other members share the same memory space. This is the biggest difference between structure and union.
	
	union student
	{
	int rollno;
	char gender;
	float marks;
	}s1;
	
	In above example variable marks is of float type and have largest size (4 bytes). So the total memory required to store union variable s1 is 4 bytes.

Difference b/w Structure & Union continue…

Pictorial Representation

Structure	Union
 s1 2 bytes 1 byte 4 bytes rollno gender marks 7 bytes	 s1 rollno (2 bytes) gender (1 byte) marks (4 bytes)

Difference b/w Structure & Union cont…

Structure	Union

We can access any member in any sequence.
 	
s1.rollno = 20;
s1.marks = 90.0;
printf(“%d”,s1.rollno);
	We can access only that variable whose value is recently stored.
	
	s1.rollno = 20;
	s1.marks = 90.0;
	printf(“%d”,s1.rollno);
	
	The above code will show erroneous output. The value of rollno is lost as most recently we have stored value in marks. This is because all the members share same memory space.

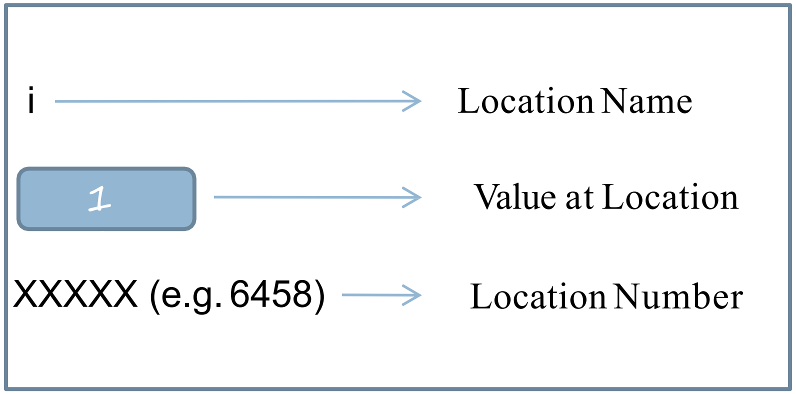
All the members canbe initialized while declaring the variable of structure.	Only first member can be initialized while declaring the variable of union. In above example, we can initialize only variable rollno at the time of declaration of variable.

Pointers

int i = 10;
What the declaration tells the C
mpil

- ❑ Reserve space in memory to hold the integer value
- ❑ Associate the name **i** with this memory location
- ❑ Store the value 10 at this location

Memory Map



Code Snippet

```
#include <stdio.h>

int main()
{
  int intVariable = 10;
  printf("Address of intVariable = %u", &intVariable);
 printf("Value of intVariable = %d", intVariable);
 return 0;
}
```

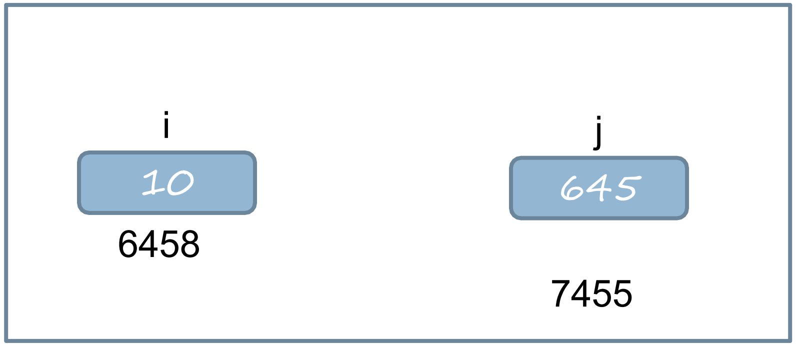
Apointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is− **type *var-name;**

Pointers cont…

Code Snippet 1

```
#include <stdio.h>

int main()
{
  int i = 10;
  int *j;
  j = &i;
 return 0;
}
```

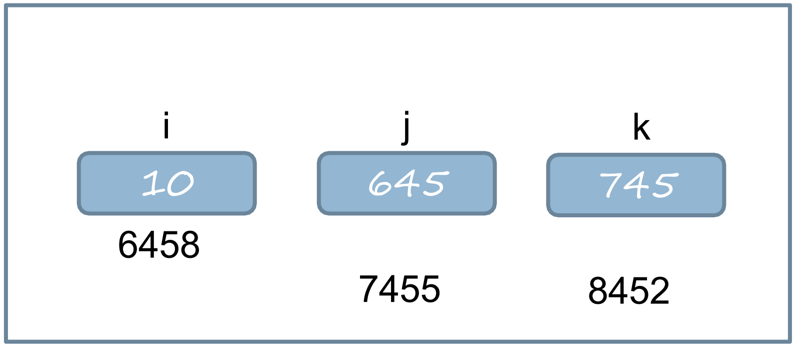
Memory Map



Code Snippet 2

```
#include <stdio.h>

int main()
{
  int i = 10, *j, **k;
  j = &i;
  k = &j;
 return 0;
}
```

Memory Map



Pointers cont… NULL Pointers

It is always a good practice to assign a **NULL** value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The **NULL** pointer is a constant with a value of zero defined in several standard libraries.

Code Snippet

```
#include <stdio.h>
int main()
{
   int *ptr = NULL;
   printf("The value of ptr is : %u\n", ptr );
   return 0;
}
```

Null Pointer Checking

1. if (ptr == null) or if (ptr <> null)
2. If (ptr) or if (!ptr)

Commonly done mistakes

```
int c, *pc;
pc=c;  /* pc is address whereas, c is not an address. */
*pc=&c; /* &c is address whereas, *pc is not an address. */
```

Dynamic Memory Allocation

The exact size of array is **unknown** until the **compile time** i.e. time when a compiler compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes **insufficient** and sometimes **more than required**.

What?

The process of allocating memory during program execution is called dynamic memory allocation. It also allows a program to obtain more memory space, while running or to release space when no space is required.

Difference between static and dynamic memory allocation

Sr #	Static Memory Allocation	Dynamic Memory Allocation
1	User requested memory will be allocated at compile time that sometimes insufficient and sometimes more than required.	Memory is allocated while executing the program.
2	Memory size can"t be modified while execution.	Memory size can be modified while execution.

Dynamic Memory Allocation cont…

Functions available in C for memory management

Sr #	Function	Description
1	void *calloc(int num, int size)	Allocates an array of num elements each of which size in bytes will be size.
2	void free(void *address)	Releases a block of memory block specified by address.
3	void *malloc(int num)	Allocates an array of num bytes and leave them initialized.
4	void *realloc(void *address, int newsize)	Re-allocates memory extending it up to new size.

Dynamic Memory Allocation Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int *pi;
  float *pj;
  pi = (int *) malloc(sizeof(int));
  pj = (float *) malloc(sizeof(float));
  *pi = 10;
   *pj = 3.56;
   printf(“integer = %d, float = %f”, *pi, *pj);
   free(pi);
   free(pj);
   return 0;
}
```

IMH

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int *pi;
  float  *pj;
  pi = (int *) malloc(sizeof(int));
  pj = (float *) malloc(sizeof(float));
  if (!pi || !pj)
  {
     printf(“Insufficient Memory”);
     return;
  }
  *pi = 10;
   *pj = 3.56;
   printf(“integer = %d, float = %f”, *pi, *pj);
   free(pi);
   free(pj);
  return 0;
}
```

IMH : Insufficient Memory Handling

DMArealloc Example

CProgram illustrating the usage of realloc

```
#include <stdio.h>
#include<stdlib.h>
int main()
{

    char *mem_allocation;
  /*  memory is allocated dynamically */
  mem_allocation = malloc( 20 * sizeof(char) );
  if( mem_allocation== NULL )
  {
     printf("Couldn't able to allocate requested memory\n"); return;
  }
  else
  {
    strcpy( mem_allocation,“dynamic memory allocation for realloc function");
  }
  printf("Dynamically allocated memory content  : %s\n", mem_allocation );
 mem_allocation=realloc(mem_allocation,100*sizeof(char));
  if( mem_allocation== NULL )
  {
     printf("Couldn't able to allocate requested memory\n");
  }
  else
  {
     strcpy( mem_allocation,"space is extended upto 100 characters");
  }
  printf("Resized memory : %s\n", mem_allocation );
  free(mem_allocation);
  return 0;
}
```

Difference between calloc and malloc

Sr #	malloc	calloc
1	It allocates only single block of requested memory	It allocates multiple blocks of requested memory
2	doesn‟t initializes the allocated memory. It contains garbage values.	initializes the allocated memory to zero

3	int *ptr;
	ptr = malloc(20 * sizeof(int));
	
	For the above, 20*2 bytes of memory only allocated in one block.
	
	Total = 40 bytes
		int *ptr;
		Ptr = calloc(20, 20 * sizeof(int));
		
		For the above, 20 blocks of memory will be created and each contains 20*2 bytes of memory.
		
		Total = 800 bytes

Algorithm Specification

An algorithm is a **finite** set of instructions that, if followed, **accomplishes a particular task**. In addition, all algorithms must satisfy the following criteria:

1. **Input**. There are zero or more quantities that are externally supplied.
2. **Output**. At least one quantity isproduced.
3. **Definiteness**. Each instruction is clear and unambiguous.
4. **Finiteness**. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness**. Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible.

Difference between an algorithm & program – program does not have to satisfy the 4th condition.

Describing Algorithm

1. **Natural Language –**e.g. English, Chinese - Instructions must be definite and effectiveness.
2. **Graphics representation –e.g. Flowchart -** work well only if the algorithm is small and simple.
3. **Pseudo Language -**
   - Readable
   - Instructions must be definite and effectiveness
4. **Combining English and C**

Algorithm Specification cont…

Describing Algorithm – Natural Language

Problem - Design an algorithm to add two numbers and display result.

```
Step 1 – START
Step 2 – declare three integers a, b & c
Step 3 – define values of a & b
Step 4 – add values of a & b
Step 5 – store output of step 4 to c
Step 6 – print c
Step 7 – STOP
```

Note - Writing step numbers, is optional.

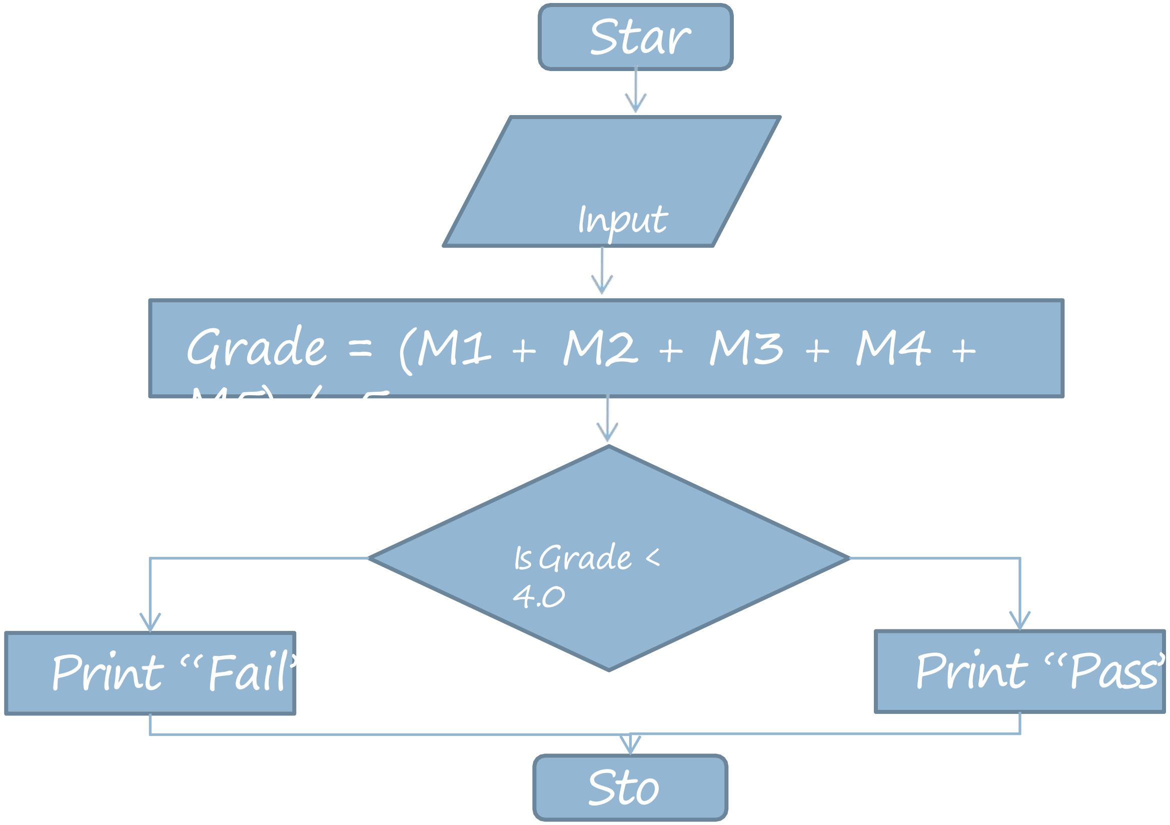
Problem - Design an algorithm find the largest data value of a set of given positive data values

```
Step 1 – START
Step 2 – input NUM
Step 3 – LARGE = NUM
Step 4 – While (NUM >=0)
            if (NUM > LARGE) then
               LARGE = NUM
            input NUM
Step 5 – display "Largest Value is:", LARGE
Step 6 – STOP
```

Algorithm Specification cont…

Describing Algorithm – Flowchart

Problem –Write an algorithm to determine a student’s final grade and indicate whether it is passing of failing. The final grade is calculated as the average of five marks.



Pseudo Language

The Pseudo language is neither an algorithm nor a program. It is an abstract form of a program. It consists of English like statements which perform the specific operations. It employs **programming-like** statements to depict the algorithm and no standard format (language independent) is followed. The statements are carried out in a order & followings are the commonly used statements.

Statement	Purpose	General Format	
Input	Get Information	**INPUT**: Name of variable e.g. INPUT: user_name	
Process	Perform an atomic activity	Variable ← arithmetic expression e.g. x ← 2 or x ← x + 1 or a ← b * c	

Decision	Choose between different alternatives	IF (condition is met)
		then
		statement(s)
		ENDIF
			IF (condition is met)
			THEN statement(s)
			ELSE
			statemen
			ts(s) ENDIF

		e.g.
		IF (amount < 100)
		THEN interestRate
		← .06
		ENDIF
			e.g.
			IF (amount < 100)
			THEN interestRate
			← .06
			ELSE
			interest Rate
			← .10 ENDIF

Pseudo Language cont…

Statement	Purpose	General Format	

Repetition	Perform a step multiple times	**REPEAT**
		statement(s)
		UNTIL (condition is met)
			WHILE (condition is met)
			statement(s)
			ENDWHILE

		e.g. count ← 0 REPEAT ADD 1 to count OUTPUT: count UNTIL (count < 10) OUTPUT: “The End”	**e.g.** count ← 0 WHILE (count < 10) ADD 1 to count OUTPUT: count ENDWHILE OUTPUT: “The End”
Output	Display information	**OUTPUT**: Name of variable e.g. OUTPUT: user_name	**OUTPUT**: message OUTPUT: ,,Credit Limit" limit

Pseudo Language Guidelines

Guidelines	Explanation
Write only one statement per line	Each statement in pseudo code should express just one action for the computer. If the task list is properly drawn, then in most cases each task will correspond to one line of pseudo code. **Task List** **Pseudo code** Read name, hours worked, rate of pay INPUT: name, hoursWorked, payRate gross = hours worked * rate of pay gross ← hoursWorked * payRate Write name, hours worked, gross OUTPUT: name, hoursWorked, gross
Capitalize initial keyword	In the example above note the words: **INPUT** and **OUTPUT**. These are just a few of the keywords to use, others include: **IF**, **ELSE**, **REPEAT**, **WHILE**, **UNTIL**, **ENDIF**
Indent to show hierarchy	Each design structure uses a particular indentation pattern. ❑ **Sequence** – Keep statements in sequence all starting in the same column ❑ **Selection** – Indent statements that fall inside selection structure, but not the keywords that form the selection ❑ **Loop** – Indent statements that fall inside the loop but not keywords that form the loop INPUT: name, grossPay, taxes IF (taxes > 0) net ← grossPay – taxes ELSE net ← grossPay ENDIF OUTPUT: name, net

Pseudo code Guidelines cont…

Guidelines	Explanation
End multiline structures	INPUT: name, grossPay, taxes IF (taxes > 0) net ← grossPay – taxes ELSE net ← grossPay ENDIF OUTPUT: name, net Watch the IF/ELSE/ENDIF as constructed above, the ENDIF is in line with the IF. The same applies for WHILE/ENDWHILE etc…
Keep statements language independent	Resist the urge to write in whatever language you are most comfortable with, in the long run you will save time. Remember you are describing a logic plan to develop a program, you are not programming!

Pseudo Language Example

1. **Problem -** Design the pseudo code to add two numbers and display the average.

```
INPUT: x, y
sum ← x + y
average ← sum / 2
OUTPUT: 'Average is:' average
```

2. **Problem** - Design the pseudo code to calculate & display the area of a circle

```
INPUT: radius
area← 3.14 * radius * radius
OUTPUT: 'Area of the circle is 'area
```

2. **Problem** - Design the pseudo code to calculate & display the largest among 2 numbers

```
INPUT: num1, num2
max ←num1
IF (num2 > num 1) THEN
  max ←num2
ENDIF
OUTPUT: 'Largest among 2 numbers is' max
```

Algorithm Specification cont…

Example - Translating a Problem into an Algorithm – Combining English and C

- **Problem -** Devise a program that sorts a set of n>= 1 integers
- Step I – **Concept** – looks good but not an algorithm
  - ✓ From those integers that are currently unsorted, find the smallest and place it next in the sorted list – selection sort
- Step II – An **algorithm**, written in Cand English

```
for (i= 0; i< n; i++)
  {
     Examine list[i] to list[n-1] and suppose that the smallest integer is list[min];
     Interchange list[i] and list[min];
  }
```

- Optional Step III – **Coding**– translating the algorithm to Cprogram

```
void sort(int *a, int n) {
     for (i= 0; i< n; i++) {
      int j=i;
      for (int k= i+1; k< n; k++)
        { if (a[k]< a[j]) {
          j = k; int temp=a[i]; a[i]=a[j]; a[j]=temp;
         }
        }
      }
  }
```

Algorithm Specification cont…

Translating a Problem into an Algorithm cont… Correctness Proof

- Theorem
  - ✓ Function sort(a, n) correctly sorts a set of n>= 1 integers. The result remains in a[0], ..., a[n-1] such that a[0]<= a[1]<=...<=a[n-1].
- Proof

  For i= q, following the execution of lines, we have a[q]<= a[r], q< r< =n-1.

  For i> q, observing, a[0], ..., a[q] are unchanged.

  Hence, increasing i, for i= n-2, we have a[0]<= a[1]<= ...<=a[n-1]

Recursive Algorithm

Arecursive algorithm is an algorithm which calls itself. In general, recursive computer programs require more memory and computation compared with iterative algorithms, but they are simpler and for many cases a natural way of thinking about the problem. There are 2 types of recursive functions.

Direct recursion Function	Indirect recursion Function
Functions call themselves e.g. function α calls α	Functions call other functions that invoke the calling function again e.g. a function α calls a function β that in turn calls the original function α.

Example –
int fibo (int n)
{
if (n==1 || n==2)
return 1;
else
return (fibo(n-1)+fibo(n-2));
}
	Example –
	int func1(int n)
	{
	if (n<=1)
	return 1;
	else
	return func2(n);
	}
	
	int func2(int n)
	{
	return func1(n);
	}

Recursive Algorithm cont…

- When is **recursion an appropriate mechanism**?
  - ✓ The problem itself is defined recursively
  - ✓ Statements: if-else and while can be written recursively
  - ✓ Art of programming
- **Why recursive algorithms** ?
  - ✓ Powerful, express an complex process very clearly
- **Properties** - A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have –
  - ✓ **Base criteria** – There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
  - ✓ **Progressive criteria**– The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.
- **Implementation** - Many programming languages implement recursion by means of **stack.**

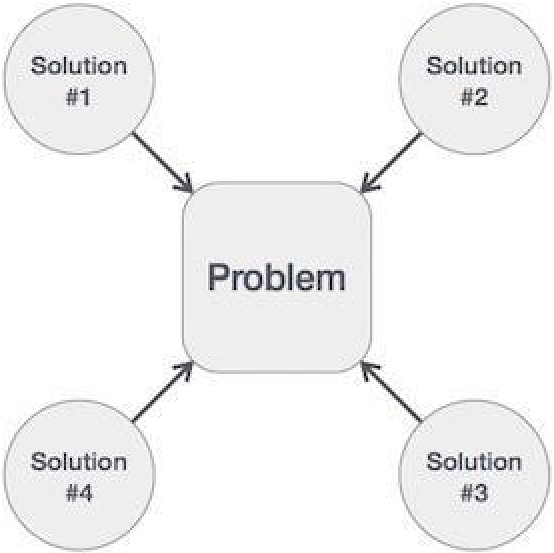
Recursive Implementation of Fibonacci

```
#include<stdio.h>
void Fibonacci(int);
int main()
{
   int k,n;
   long int i=0,j=1,f;

   printf("Enter the range of the Fibonacci series: ");
   scanf("%d",&n);

   printf("Fibonacci Series: ");
   printf("%d %d ",0,1);
   Fibonacci(n);
   return 0;
}
```

```
//continuation of program
void Fibonacci(int n)
{
   static long int first=0,second=1,sum;

   if(n>0)          Base Criteria
   {
       sum = first + second;
       first = second;
       second = sum;
       printf("%ld ",sum);
       Fibonacci(n-1);  Progressive Criteria
   }
}
```

Algorithm Analysis

We design an algorithm to get solution of a given problem. Aproblem can be solved in more than one ways.



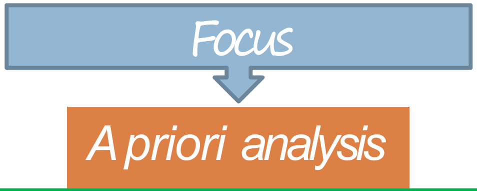
Hence, many solution algorithms can be derived for a given problem. Next step is to analyze those proposed solution algorithms and implement the best suitable.

Algorithm Analysis cont…

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation, as mentioned below −

- **Apriori analysis** − This is **theoretical analysis** of an algorithm. Efficiency of algorithm is measured by assuming that all other factors e.g. processor speed, are constant and have no effect on implementation.

- **A posterior analysis** − This is **empirical analysis** (by means of direct and indirect observation or experience) of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required and are collected.



Focus

A priori analysis

Algorithm Complexity

Suppose Xis an algorithm and n is the size of input data, the time and space used by the Algorithm Xare the two main factors which decide the efficiency of X.

- ❑ **Time Factor –** The time is measured by counting the number of key operations such as comparisons in sorting algorithm

- ❑ **Space Factor –** The space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and / or storage space required by the algorithm in terms of n as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. Space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example simple variables & constant used, program size etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example dynamic memory allocation, recursion stack space etc.

Space complexity SC(P) of any algorithm **P** is SC(P) = FP + VP (I) where FP is the fixed part and VP(I) is the variable part of the algorithm which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

```
Algorithm: SUM(A, B)
Step 1 - START
Step 2 - C←A+ B+10
Step 3 - Stop
```

3 variables and data type of each variable is int, So VP(3) = 3*2 = 6 bytes (No. of characteristic i.e. I = 3)
1 constant (i.e. 10) and it is int, so FP = 1*2 = 2 bytes
So - SC(SUM) = FP + VP (3) = 2 + 6 = 8 bytes

Space Complexity cont…

Example 1

```
int square(int a)
{
   return a*a;
}
```

In above piece of code, it requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for return value.

That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be **Constant Space Complexity**.

If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity

Example 2

```
int sum(int a[], int n)
{
  int sum = 0, i;
  for(i = 0; i < n; i++) { sum = sum + A[i];}
  return sum;
}
```

In above piece of code it requires -

- 2*n bytes of memory to store array variable ‘a[]’
- 2 bytes of memory for integer parameter 'n‘
- 4 bytes of memory for local integer variables 'sum' and 'i' (2 bytes each)
- 2 bytes of memory for return value.

That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the amount of memory depends on the input value of 'n'. This space complexity is said to be **Linear Space Complexity.**

If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear Space Complexity

Time Complexity

Time Complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as **number of steps * time taken by each steps**

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is **T(n) = c*n**, where c is the time taken for addition of two bits. Here, we observe that T(n) grows linearly as input size increases.

Asymptotic analysis

Asymptotic analysis of an algorithm, refers to defining the mathematical foundation/framing of its run-time performance.

Asymptotic analysis are input bound i.e., if there's no input to the algorithm it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Time Complexity cont… Asymptotic analysis

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.

For example,

- Running time of one operation is computed as **f(n)**
- May be for another operation it is computed as **g(n^2)**

Usually, time required by an algorithm falls under three types –

❑ **Best Case** – Minimum time required for program execution.

❑ **Average Case** – Average time required for program execution.

❑ **Worst Case** – Maximum time required for program execution.

Time Complexity cont… Best, Average and Worst Case examples

Example 1: Travel from Kanyakumari to Srinagar

- ❑ **Worst Case** – You go to Ahemdabad and then you take a east go to Guhwati and then again take a west and go to Srinagar.
- ❑ **Average Case** – You go normal route highways only to reach there.
- ❑**Best Case** – Take every short cut possible leave the national highway if you need to take state highway, district road if you need to but you have to get to Srinagar with least possible distance

Example 2: Sequential Search for k in an array of nintegers

- ❑ **Best Case** – The 1st item of the array equals to k
- ❑ **Average Case** – Match at n/2
- ❑ **Worst Case** – The last position of the array equals to k

Time Complexity cont…Asymptotic Notations

Following are commonly used asymptotic notations used in calculating running time complexity of an algorithm.

- **O Notation –"Big Oh"** - The O(n) is the formal way to **express the upper bound** of an algorithm's running time. It **measures the worst case time complexity or longest amount of time** an algorithm can possibly take to complete.

- **Ω Notation –"Omega"** - The Ω(n) is the formal way to **express the lower bound** of an algorithm's running time. It **measures the best case time complexity or best amount of time** an algorithm can possibly take to complete.

- **θ Notation –"Theta" -** The θ(n) is the formal way to **express both the lower bound and upper bound** of an algorithm's running time.

Time Complexity cont…Asymptotic Notations cont…

Sr #	Algorithm	Time Complexity		
		Best	Average	Worst
1	Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
2	Insertion Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
3	Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$
4	Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
5	Heap Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$

Time-Space Trade Off

The best algorithm to solve a given problem is one that requires less space in memory and takes less time to complete its execution. But in practice it is not always possible to achieve both these objectives. As we know there may be more than one approach to solve a particular problem. One approach may take more space but takes less time to complete its execution while the other approach may take less space but takes more time to complete its execution. We may have to sacrifice one at the cost of the other. If space is our constraint, then we have to choose a program that requires less space at the cost of more execution time. On the other hand if time is our constraint then we have to choose a program that takes less time to complete its execution at the cost of more space.

Time Complexity Rules

- Time complexity of a function (or set of statements) is considered as O(1) if it doesn’t contain **loop**, **recursion** and **call to any other non-constant time function**. **Example:**

```
void swap(int *x, int *y)
{
   int temp;
   temp=*x;
   *x=*y;
   *y=temp;
}
```

Time complexity of swap function= Total number of simple statements = 4 = constant = **O(1)**

Note -

Constant Time - algorithm runs in a fixed amount of time, it just means that it isn't proportional to the length/size/magnitude of the input. i.e., for any input, it can be computed in the same amount of time (even if that amount of time is really long).

Time Complexity Rules for Loops

- A loop or recursion that runs a constant number of times is also considered as **O(1). Example -**

```
int i;
for (i = 1; i <= c; i++) // c is a constant
{
   // some O(1) expressions
}
```

- Time Complexity of a loop is considered as **O(n)** if the loop variables i is incremented / decremented by a constant amount c. **Example -**

```
for (i = 1; i <= n; i = i+c)
{
  // some O(1) expressions
}
```

Time Complexity Rules for Loops cont…

- Time Complexity of a loop is considered as **O(log n)** if the loop variables i is multiplied or divided by a constant amount c. **Example** -

```
for (i = 1; i <= n; i=i*c)
{
  // some O(1) expressions
}
```

- Time Complexity of a loop is considered as **O(log log n)** if the loop variables i if the loop variables is reduced / increased exponentially by a constant amount c. **Example**-

Example 1	Example 2

Here c is a constant greater than 1 for (i = 2; i <= n; i=pow(i,c))
{
// some O(1) expressions
}
	//Here fun is sqrt or cuberoot or any other constant root for (i = n; i > 0; i = fun(i))
	{
	// some O(1) expressions
	}

Time Complexity Rules for Loops cont…

- Time complexity of nested loops is equal to the number of times the innermost statement is executed that is nothing but the multiplication of outer loop complexity into inner loop complexity.

Sr #	Program Segment	Time Complexity	Explanation

1	int i, j, k;
	for (i = 1; i <=m; i ++)
	{
	for (j = 1; j <=n; j ++)
	{
	k=k+1;
	}
	}
		$O(m*n)$
		If m = n, then $O(n^2)$
			Outer for m times and inner for n times, so total m*n times

2	int i, j, k;
	for (i = n; i >1; i =i-2)
	{
	for (j = 1; j <=n; j=j+3)
	{
	k=k+1;
	}
	}
		$O(n^2)$	Outer loop approx n times and inner approx n times, so total n^2 times.

Time Complexity Rules for Loops cont…

Sr #	Program Segment	Time Complexity	Explanation

3	int i, j, k;
	for (i = n; i >1; i =i*2)
	{
	 for (j = 1; j <=n; j++)
	 {
	  k=k+1;
	 }
	}
		O(n log n)	Outer loop approx log n times, inner loop n times, so total n log n times.

4	int i, j, k;
	for (i = 1; i<=n; i=i*2)
	{
	 for (j =n; j>=1; j=j/2)
	 {
	  k = k +1;
	 }
	}
		O((log n)^2)	Outer loop approx log n times and inner loop approx log n times, so total ((log n)^2 times.

Time Complexity Rules for Loops cont…

Sr #	Program Segment	Time Complexity	Explanation

5	for (i = 1; i<=n; i++)
	{
	for (j =1; j<=100; j++)
	{
	Simple-Statements;
	}
	}
		O(n)	The outer loop executes approx. n times, and the innermost 100 times. Here n=O(n) and 100=O(1) constant time. So total = O(n)*O(1) = O(n)

- When there are consecutive loops, the time complexity is calculated as sum of time complexities of individual loops and final time complexity is the higher order term in terms of n (the one which is larger than others for larger value of n)

Sr #	Program Segment	Time Complexity	Explanation

1	for (int i = 1; i <=m; i += c)
	{
	// some O(1) expressions
	}
	for (int i = 1; i <=n; i += c)
	{
	// some O(1) expressions
	}
		O(m+n)
		If m=n, then O(2n)=O(n)
			First for outer i loop O(m), second for inner i loop O(n) times, so total O(m+n) times

Time Complexity Rules for Loops cont…

$1<logn< n <n<nlogn<n^2<n^3<....<2^n<3^n<...<n^n$

Sr #	Program Segment	Time Complexity	Explanation

2	for (int i = 1; i <=n; i *= 2)
	{
	  // some O(1) expressions
	}
	for (int i = 1; i <=n; i ++)
	{
	  // some O(1) expressions
	}
		O(n)	First for outer i log n times, second inner i it is n times. Now total=log n + n = O(n) as n is asymptotically larger than log n.

3	int i, j, k,l;
	for (i = 1; i <=n; i ++)
	{
	  for (j = 1; j <=n; j=j*2) {p=i+j;}
	}
	
	for (k = n; k >=1; k=k/3)
	{ q=k+p; }
	
	for (l = n; l >=1; l=l-2)
	{ q=k+p; }
		O(n log n)	Nested for-ij loop is executed n log n times. k loop log n times, l loop n times. So total= n log n + log n + n = O(n log n) as n log n > n > log n

Time Complexity Rules for Recursion

Sr #	Program Segment	Time Complexity	Explanation

1	**Factorial of a number**
	int Factorial (int n)
	{
	if n == 0
	then
	return 0;
	else
	return n * Factorial(n – 1)
	}
		$O(n)$	comparison, multiplication and subtraction are simple statements and each takes 1 unit of time. So $T(n) = T(n-1) + 3$ if $n > 0$ $T(0) = 1$ if $n = 0$
			So $T(n) = T(n-1) + 3$
			$= T(n-2) + 6$
			$= T(n-k) + 3k$
			When $k = n$, $T(n) = T(0) + 3n$
			$= 3n + 1$
			So $T(n) = O(n)$

2	**Fibonacci Sequence**
	int Fib(int n)
	{
	if n <= 1 then
	return
	1; else
	return Fib(n-1) + Fib (n – 2)
	}
		??	??

Space Complexity Rules

Sr #	Program Segment	Space Complexity	Explanation

1	int sum(int x, int y, int z)
	{
	int r = x + y
	+ z; return r;
	}
		O(1)	requires 3 units of space for the parameters and 1 for the local variable, and this never changes, so this is O(1).

2	int sum(int a[],
	int n) { int r =
	0;
	for (int i = 0; i < n; ++i) {
	r += a[i];
	}
	return r;
	}
		O(n)	requires n units for a, plus space for n, r and i, so it's O(n).

3	void matrixAdd(int a[], int b[], int
	c[], int n) { for (int i = 0; i < n;
	++i) {
	c[i] = a[i] + b[j]
	}
	}
		O(n)	requires n units for a, b and c plus space for n, and i, so it's O(n).

Summary

Detailed Lessons

Functions, Structures and Unions, Pointers, Dynamic Memory Allocation, Algorithm Specification, Space and Time Complexity

How was the journey?

Find time complexity

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
   a = a + rand();
}
for (j = 0; j < M; j++) {
   b = b + rand();
}
```

Find time complexity

```
int a = 0;
for (i = 0; i < N; i++) {
   for (j = N; j > i; j--) {
      a = a + i + j;
   }
}
```

THANK YOU!

Assignments

1. Find the time complexity of the following code segment
   - ✓
     ```
     for (i = 1; i <= n; i = i*2)
      {
        for (j = n; j >= 1; j = j/2)  { some statement }
      }
     ```
   - ✓
     ```
     for (j= c; j > 0; j--)   // consider c as const
     {
       some statement
     }
     ```
   - ✓
     ```
     for (i = n; i <= 1; i = i/c) // consider c as const
      {
        // some statement
      }
     ```
2. Write an recursive algorithm to print from 1 to n (where n > 1)
3. Write an recursive algorithm to find the kth smallest element of a set S
4. Write an recursive algorithm to sum the list of numbers

Assignments

5. Find the space complexity of the following code segment

```
int Factorial (int n)
      {
        if n== 0 then
          return 0;
        else
          return n * Factorial(n – 1)
      }
```

6. Find the time and space complexity of the following code segment

✓ int Fib(int n)

```
      {
        if n <= 1 then
          return 1;
        else
          return Fib(n-1) + Fib (n – 2)
      }
```

Home Work (HW)

- Find the time and space complexity of the following code segment
  - ✓

```
int GCD(int x, int y)
 {
   if y == 0 then
     return x
   else if x >= y AND y > 0
    return GCD(y, x % y)
    else
      return 0;
 }
```

Home Work (HW)

- Find the time and space complexity of the following code segment

```
void matrixAddition(int a[][], int b[][], int c[][], int n)
 {
  int i,j;
  for (i = 0; i < n; ++i)
  {
   for (j = 0; j < n; ++j)
   {
    c[i][j] = a[i][j] + b[i][j];
   }
  }
 }
```

Supplementary Reading

- ❑ Watch the following video
  - ✓ https://www.youtube.com/watch?v=8syQKTdgdzc
  - ✓ https://www.youtube.com/watch?v=AL7yO-I5kFU
  - ✓ http://nptel.ac.in/courses/106102064/1
- ❑ Read the following
  - ✓ https://www.tutorialspoint.com/data_structures_algorithms/

FAQ

- **What is Information** - If we arrange some data in an appropriate sequence, then it forms a Structure and gives us a meaning. This meaning is called Information . The basic unit of Information in Computer Science is a bit, Binary Digit. So, we found two things in Information: One is Data and the other is Structure.
- **What is Data Structure?**
  1. A data structure is a systematic way of organizing and accessing data.
  2. A data structure tries to structure data!
     - Usually more than one piece of data
     - Should define legal operations on the data
     - The data might be grouped together (e.g. in an linked list)
  3. When we define a data structure we are in fact creating a new data type of our own.
     - i.e. using predefined types or previously user defined types.
  - Such new types are then used to reference variables type within a program

FAQcont…

- **Why Data Structures?**
  1. Data structures study how data are stored in a computer so that operations can be implemented efficiently
  2. Data structures are especially important when you have a large amount of information
  3. Conceptual and concrete ways to organize data for efficient storage and manipulation.
- **ADT**
  1. Abstract Data Types (ADT's) are a model used to understand the design of a data structure
  2. 'Abstract' implies that we give an implementation-independent view of the data structure
  3. ADTs specify the type of data stored and the operations that support the data
  4. Viewing a data structure as an ADT allows a programmer to focus on an idealized model of the data and its operations

FAQcont…

- **Time Complexity of algorithm -** Time complexity of an algorithm signifies the total time required by the program to run till its completion. The time complexity of algorithms is most commonly expressed using the **big O** notation. Time Complexity is most commonly estimated by counting the number of elementary functions performed by the algorithm. And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the worst-case Time complexity of an algorithm because that is the maximum time taken for any input size.
- **Types of Notations for Time Complexity**
  1. Big Oh denotes "fewer than or the same as" <expression> iterations.
  2. Big Omega denotes "more than or the same as" <expression> iterations.
  3. Big Theta denotes "the same as" <expression> iterations.
  4. Little Oh denotes "fewer than" <expression> iterations.
  5. Little Omega denotes "more than" <expression> iterations. idealized model of the data and its operations