UNIT 3

Pointers

* A pointer is a variable which contains the address in memory of another variable.

* Pointers are mainly used to dynamically allocate and deallocate memory, especially when we need to handle unlimited amonts of data.

* Pointers are used in C program to access the memory and manipulate the address.

* memory is accessed efficiently with pointers

* It helps to allocate and deallocate memory at run time.

* It provides a way to return multiple data items from a function using its function arguments.

* It helps to write an efficient program.

* It directly deals with hardware components

* It also provides an alternate way to access an array element

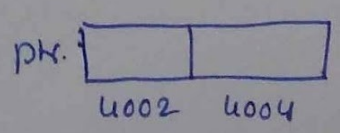
* It helps to directly access the memory location.

Declaring, initializing & Accessing of pointer variables :

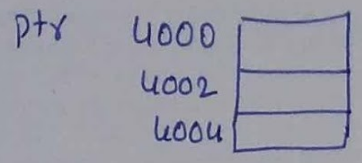
→ An integer pointers points to an integer variable

→ A character pointer points to a character variable

int *ptr → ptr is declared



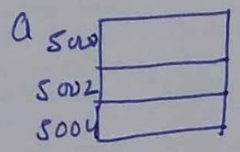
To make 'ptr' is declared, it is given a memory location (i.e) 4000.



To make 'ptr' to point an integer variable, declare an integer variable

int a = 100;

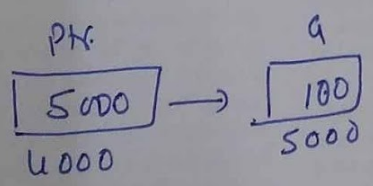
When it is declared, a memory location for 'a' is given (i.e) 5000.



To assign the address (&) of 'a' to 'ptr'

ptr = &a;

The result of assigning address of 'a' to 'ptr' is shown below:



Thus * and & are pointer operators. *ptr represents the value of the variable pointed by the pointer.

Declaration of pointers of different data types:-

```
int *p;
char *c;
float *f;
```

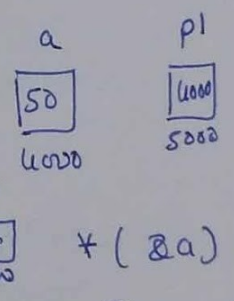
The following shows the declaration and use of void pointer.

```
void *p;    // declares a void pointer
float x=30;
p= (float *) &x;
```

The void pointer is a generic pointer. It can be used to point to any type of variable. Type casting is needed before using the variable 'p'.

Accessing a variable through its pointer:- Once a variable has been pointed by a pointer it can be accessed through the pointer.

```
int a = 50;
int *p1 = &a;
*p1 = 20;
```



$*(\&a) = 20$

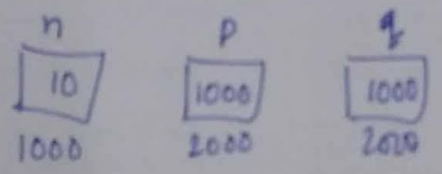
$\Rightarrow a = 20.$

Pointer operations:-

(i) Pointer assignments
(ii) Pointer conversion
(iii) Pointer arithmetic
(iv) Pointer comparison
(v) Pointer expression.

(i) **Pointer assignments:-** Pointer assignment between two pointers makes them point to the same variable. So the assignment $y=x$ makes y point to the same variable as x.

```
void main()
{
  int n=10, *p, *q;
    p = &n;
    q = p   // both p and q points to n
  Pf ("%d %d %d", n, *p, *q);   ⇒  10, 10, 10
  Pf ("%d %d %d", &n, p, q);   ⇒  1000 1000 1000
  getch();
}
```



<u>Pointer Conversion</u>:- A pointer to one type of value can be converted to a pointer to a different type.

* A pointer to void can be converted to or ~~from~~ from a pointer to any type without restriction or loss of information. If the result is converted back to the original type, the original pointer is ~~retored~~. recovered

The casting can be done using void or other types of pointers

1. Casting of void * pointer
2. Casting of other types of pointers → produce wrong result

ex
```
float f=2.3, *p1;
int *p2;
void *p3;
p1 = &f   ✓
p3 = (float *) &f;   ✓
p2 = (int *) &f;   ✗ wrong result.
```

```
#include <stdio.h>
void main()
{
    int *p1, *p2;
    int n, i = 2;
    int m = {1,2,3,4,5,6,7,8,9,10};
    clrscr();
    p2 = &m[9];
    p1 = m;
    Pf("\n%p \t %d", p1, *p1);
    p1 = p1 + 5;
    Pf("\n %p \t %d", p1, *p1);
    p1++;
    Pf("\n %p \t %d", p1, *p1);
    p1--;
    Pf("\n %p \t %d", p1, *p1);
    p1 = p1-5;
    Pf("\n %p \t %d", p1, *p1);
    Pf("\n %p \t %d", p2, *p2);
    n = p2 - p1;
    Pf("\n No of elements : %d", n);
    p2 = p2-i;
    Pf("\n %p \t %d", p2, *p2);
}
```

(iii) Pointer arithmetic :- pointer is an address, which is a numeric value.

we can perform ~~arit~~ arithmetic operations on a pointer just ~~as~~ we do on a numeric value. ++, --, + and -

```
int *q;
q++ / ++q  → increment 'q' 2 bytes

float *r;
r++ / ++r  → increament 'r' 4 bytes

char *s;
s++ / ++s  → increament 's' 1 byte
```

Addition Arithmetic with pointers:

* Adding an address with an integer gives an address.
  - ⇒ Address + Integer → Address
  - ⇒ Address++ → Address
  - ⇒ Address - Integer → Address
  - ⇒ Address-- → Address
  - ⇒ ++ Address → Address
  - ⇒ -- Address → Address
  - ⇒ Address - Address → number

illegal Arithmetic with pointers:-

Address + Address ⇒ illegal
Address * Address → illegal
Address / Address → illegal
Address % Address → illegal

(iv) Pointer Comparison:-

Comparison of pointers of same type

```
#include <stdio.h>
void main()
{
    int *ptr1, *ptr2;
    int a, b;
    ptr1 = &a;
    ptr2 = &b;
  if (ptr2 > ptr1)
    pf("ptr is far from ptr1");
}
```

Pointer Comparison of different data types:

```
#include <stdio.h>
void main()
{ int *ptr1;
  float *ptr2;
  int a;
  float f;
  ptr1 = &a;
  ptr2 = &f;
  if (ptr2 > ptr1)
    pf("ptr2 is far from ptr1");
}
```

* Two pointers of different data types can be compared.
* Pointers can store address of any data type, address of the data type is 'integer' so we can compare address of any two pointer although they are of different data types.

$>, <, >=, <=, ==, !=$

Ex

```
#include <stdio.h>
#include <conio.h>
void main()
{ int a=4, b=2, *ap, *bp, *sp;
  int s, d, p, q, r, t;
  ap = &a;
  bp = &b;
  s = *ap + *bp;
  d = *ap - *bp
  p = *ap * *bp
  q = *ap / *bp;
  r = *ap % *bp;
  sp = &t;
  *sp = *ap + *bp;
  pf("\n %d %d %d %d %d %d %d %d", s, d
      p, q, r, sp);
}
```

Pointer to array:- Array is a pointed to &array[0], which is the address of the first element of the array.

```
int *p;
int array[10];
p = array;
```

~~* It is legal to use array of first element in p, you can access array elements using *p, *(p+1), *~~

* It is legal to use array names as constant pointers, and vice versa.

Once you store the address of first element in p, you can access array elements using *p, *(p+1), *(p+2) and so on.

```
array[5] = {10, 20, 30, 40, 50};
int *p;
int i;
p = array; => &array[0]
```

array[0]	[1]	[2]	[3]	[4]
10	20	30	40	50
100	102	104	106	108

*(p+0) => *(100+0) = 10

*(p+1) => *(100+1) = 20
address increment

*(array+0) => 10

*(array+4) => 50

=> &array[1] => (array+1) => *(array+1) => array[1]

* &array[i] => (array+i)

array[i] => *(array+i)

ex Program to find the sum of six numbers with arrays and pointers

```
#include <stdio.h>
void main()
{ int i, array[5] sum=0;
  Pf ("Enter 5 numbers : \n");
  for (i=0; i<5; ++i)
  {
     Sf ("%d", (array+i));
     sum += * (array+i);
  }
  Pf ("sum =%d", sum);
}
```

O/p Enter 5 no.
2
3
4
5
6
sum = 20

Pointers and strings :-

```
char name[] = "ram";
```

name [0]	[1]	[2]	
r	a	m	\0

```
char name[10] = "ram";
```

0	1	2							
r	a	m	\0						

where last 6 array elements are currently unused.

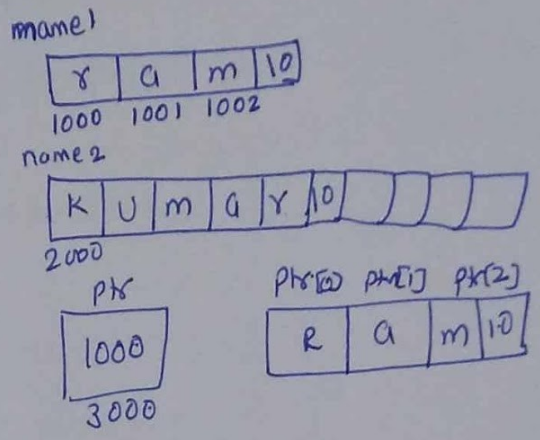
* Since these strings are really just arrays, we can access each character in the array using subscript notation.

* We need to state the maximum number of characters we will ever store in an array. This is called static allocation, where the size of the array is determined at compilation time.

String as Pointer :-

The pointers only hold an address, they cannot hold all the characters in a character array.

```
Char name1[] = "ram";
char name2[0] = "Kumar";
char *ptr;
ptr = name1;
```



ex. find the length of a string

```
#include <stdio.h>
void main()
{
  int i=0, len=0;
  char *str;
  Pf("\n Enter a string");
  gets (str);
  ~~while ( *str != '\0')~~
  while ( *str != '\0')
    {
      len++;
      i++;
    }
  Pf("\n The length of the given string is %d", len);
}
```

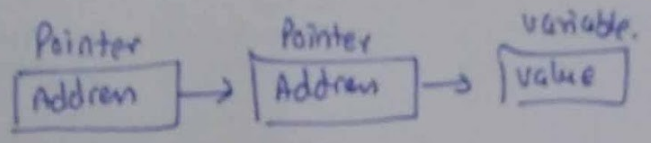
O/P

Enter the string
Hello

The length of the given string is 5.

~~Pointer to structures~~:-

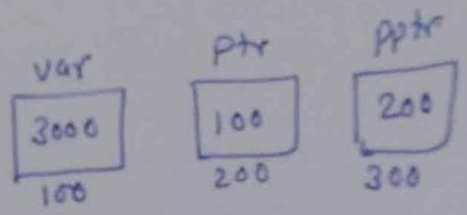
Pointer to Pointer:- A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



declaration ~~declares~~ a of pointer to a pointer :-

int **var; → pointer to pointer of type int

```
#include <stdio.h>
int main() {
{
    int var;
    int *ptr;
    int **pptr;
    var = 3000;
/* take the address of var */
    ptr = &var;
/* take the address of ptr using address of operator & */
    pptr = &ptr;
Pf("Value of var = %d \n", var);
Pf("Value of *ptr = %d \n", *ptr);
Pf("Value of **ptr = %d \n", **ptr);
return 0;
}
```



O/P

var = 3000
*pk = 3000
**pk = 3000

Null pointer:- A pointer that is assigned NULL is called null pointer.

* Null pointer is a constant with value of zero.
* It is always good practice to assign a Null value to a pointer variable in case we do not have an exact address to be assigned.

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

** most of OS, programs are not permitted to access memory at address 0 because that memory is reserved by the OS. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing

Generic Pointer:- The void pointer in C is a pointer which is not associated with any data types. It points to some data location in the storage means points to the address of variables. It is also called general purpose pointer. In C, malloc() and calloc() functions return void * or generic pointers.

It has some limitations -

1) Pointer arithmetic is not possible with void pointer due to its concrete size.
2) It can't be used as dereferenced.

ex
```
# include < stdlib.h>
int main()
{
    int a = 7;
    float b = 7.6;
    void *p;
    p = &a;
    Pf (" Integer variable is = %d", *((int *)p));
    p = &b;
    Pf (" In float variable is = %f", *( (float *)p));
    return 0;
}
```

O/P.

Integer variable is = 7
float variable is = 7.600000

Passing pointers to functions in C:- C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.

ex
```
# include <stdio.h>
# include < time.h>
void getSeconds (unsigned long *par);
int main()
{ unsigned long sec;
  getSeconds (&sec);
  Pf ("No of seconds : %ld In", sec);
  return 0;
}
```

```
void getSeconds ( unsigned long *par)
{
    *par = time (NULL);
    return;
}
```

The function, which can accept a pointer, can also accept an array.

FE
```
#include<stdio.h>
/* function declaration */
double getAverage (int *arr, int size);
int main()
{
  int balance[5] = {1000, 2, 3, 17, 50};
  double ~~avg~~; avg;
/* pass pointer to the array as an argument */
  avg = getAverage( balance, 5);
/* output the returned value */
  pf ("Average value is : %f \n", avg);
  return 0;
}
double getAverage (int *arr, int size)
{
  int i, sum = 0;
  double avg;
  for (i = 0; i<size ; ++i)
    {   sum += arr[i];
    }
  avg = (double)sum / size;
  return avg;
}
```

O/P
Average value is 214.2000

Return pointer from function in C:- C also allows to return a pointer from a function. To do so, we ~~should have~~ should have to declare a function returning a pointer as in the following example-

```
int *myfunction ()
  {

  }
```

* It is not good idea to return the address of a local variable outside the function, so we should have to define the local variable as static variable.

FE Generate 10 random numbers and return them using an array name which represents a pointer, i.e. address of first array element.

```
#include <stdio.h>
#include <time.h>
int *getRandom()
{
  static int r[10];
  int i;
/* set the seed */
srand((unsigned)time(NULL));
for (i=0; i<10; ++i)
  {  r[i]= rand();
     pf("%d \n", r[i]);
  }
return r;
}
```

```
/* main function */
int main()
{
  int *p;
  int i;
  p= getRandom();
  for (i=0; i<10; i++)
    {  pf("*(p+[%d]): %d \n", i, *(p+i));
    }
  return 0;
}
```

~~Dynamic memory allocation~~:-

Structures:- C provides a data types called 'structure'. A structure gathers together, different atoms of information that comprise a given entity.

* Ordinary variables and arrays can hold a no of pieces of information of the same data type.

But often we deal with entities that are collection of dissimilar data types

Ex we want to store data about a book

name → string no of pages → integer.

price → float

if we want to add 3 Books data then, we can follow two approaches.

(a) Construct individual arrays, one for storing names, another for storing prices and still another for storing no of pages.

(b) use a structure variable.

FE (a)

```
main()
{
  char name[3];
  float price[3];
  int page[3], i;
  Pf("\n Enter names, prices and no of pages of 3 books\n");
  for(i=0; i<=2; i++)
    Sf("%c %f %d", &name[i], &price[i], &pages[i]);
  Pf("\n And this is what you entered \n");
  for (i=0; i<=2; i++)
      Pf("%c %f %d \n", name[i], price[i], pages[i]);
}
```

O/P

```
Enter names, prices and no of pages
A 100.00 354
C 256.50 682
F 233.70 512
A 100.000000 354
C 256.50 682
F 233.70 512
```

step ④ is more difficult to handle as the number of items relating to the book go on increasing.

To solve this problem, C provides a special data type - the structure.

* A structure contains a number of data types grouped together. These data types may or may not be of the same type.

Ex

```
main ()
{
    struct book
    {
        char name;
        float price;
        int pages;
    };
    struct book b1, b2, b3;
    pf("\n Enter names, prices & no of pages of 3 books \n");
    sf("%c %f %d", &b1.name, &b1.price, &b1.pages);
    sf("%c %f %d", &b2.name, &b2.price, &b2.pages);
    sf("%c %f %d", &b3.name, &b3.price, &b3.pages);
    pf("\n This is what you entered");
    pf("\n %c %f %d", b1.name, b1.price, b1.pages);
    pf("\n %c %f %d", b2.name, b2.price, b2.pages);
    pf("\n %c %f %d", b3.name, b3.price, b3.pages);
}
```

Declaring a structure:-

```
struct <structure name>
{
  structure element 1;
  structure element 2;
  structure element 3;
  - - -
  - - -
};
```

Once the new structure data types has been defined one or more variables can be declared to be of that type.

ex struct book b1, b2, b3;

** struct book b1, b2, b3, statement sets aside space in memory. It makes available space to hold all the elements in the structure:

FF
1 byte for name
4 bytes for price
2 bytes for pages
7 bytes totals

* These bytes are always in adjacent memory locations.

types of declaration

```
struct book
{
  char name;
  float price;
  int pages;
};
struct book b1, b2, b3;
```

```
struct book
{
  char name;
  float price;
  int pages;
} b1, b2, b3;
```

```
struct ~~book~~
{
  char name;
  float price;
  int pages;
} b1, b2, b3;
```

```
struct book
{ char name[10];
  float price;
  int pages;
};
struct book b1 = {"Basic, 130.00, 550};
struct book b2 = {"physics", 150.80, 800};
```

Declaring a structure:-

```
struct <structure name>
{
  structure element 1;
  structure element 2;
  structure element 3;
  - - -
};
```

Once the new structure data types has been defined one or more variables can be declared to be of that type.

ex struct book b1, b2, b3;

** struct book b1, b2, b3, statement sets aside space in memory. It makes available space to hold all the elements in the structure:

FE
1 byte for name
4 bytes for price
2 bytes for pages
7 bytes totals

* These bytes are always in adjacent memory locations.

types of declaration

```
struct book
{
  char name;
  float price;
  int pages;
};
struct book b1, b2, b3;
```

```
struct book
{
  char name;
  float price;
  int pages;
} b1, b2, b3;
```

```
struct ~~book~~
{
  char name;
  float price;
  int pages;
} b1, b2, b3;
```

```
struct book
{ char name[10];
  float price;
  int pages;
}; struct book b1 = {"Basic, 130.00, 550};
   struct book b2 = {"physics", 150.80, 800};
```

* The closing brace in the structure type declaration must be followed by a semicolon.

* It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory. It defines the 'form' of the structure.

* Usually structure type declaration appears at the top of the source code file, before any variables or functions are defined. In very large programs they are usually put in a separate header file.

Accessing structure Elements:- structure use dot (.) operator.

* Before the dot there must always be a structure variable and after the dot there must always be a structure element.

ex b1. pages

How structure elements are stored:- whatever be the elements of a structure they are always stored in contiguous memory locations.

FE

```
main()
{
  struct book
  {
    char name;
    float price;
    int pages;
  };
  struct book b1= {'B', 130.00, 550};
  Pf("\n Address of name %u", &b1.name);
  Pf("\n Address of price %u", &b1.price);
  Pf("\n Address of pages = %u", &b1.pages);
}
```

O/P

Address of name = 65518

Address of price = 65519

Address of pages = 65523

* The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator.

ex

```
main()
{
    struct employee
    {
        char name[10];
        int age;
        float salary;
    };
    struct employee e1= {"sanjay", 30, 5500.5};
    struct employee e2, e3;

    strcpy (e2.name, e1.name);
    e2.age = e1.age;
    e2.salary= e1.salary;
    e3 = e2;

    Pf ("\n %s %d %f", e1.name, e1.age, e1.salary);
    Pf ("\n %s %d %f", e2.name, e2.age, e2.salary);
    Pf ("\n %s %d %f", e3.name, e3.age, e3.salary);

}
```

* Copying of all structure elements at one go has been possible only because the structure elements are stored in contiguous memory locations.

* One structure can be nested within another structure. Using this facility complex data types can be created.

```
main()
{
  struct address
  {
     char phone [15];
     char city [25];
     int pin;
  };
  struct emp
  {
     char name [25];
     struct address a;
  };
  struct emp e = {"jeru", "531046", "nagpur", 10};

    Pf ("\n name = %s phone = %s", e.name, e.a.phone);

    Pf ("\n city = %s pin = %d", e.a.city, e.a.pin);

  }
```

O/p

name = jeru

phone = 531046

city = nagpur

pin = 10

** The method used to access the element of a structure that is part of another structure. For this the dot operator is used twice, as in the expression.

e.a.pin or e.a.city.

* like an ordinary variable, a structure variable can also be passed to a function. We may either pass individual structure elements or the entire structure variable at one go.

Passing individual structure elements :

```
main()
{
  struct book
  {
    char name[25];
    char author[25];
    int callno;
  };
  struct book b1 = {"Let us C", "YPK", 101};
  display(b1.name, b1.author, b1.callno);
}
```

this is a mixed call by reference, call by value.

base address of array name & author

value

```
display(char *s, char *t, int n)
{
  pf("\n %s %s %d", s,t,n);
}
```

o/p

Let us C YPK 101

Passing entire structure variable at a time :

```
struct book
{
  char name[25];
  char author[25];
  int callno;
};
main()
{
  struct book b1 = {"Let us C", "YPK", 101};
  display(b1);
}
display(struct book b)
{
  printf("\n%s %s %d", b.name, b.author, b.callno);
}
```

o/p: Let us C YPK 101

Having collected what is being passed to the display() function, the question comes, how do we define the formal arguments in the function. We cannot say,

```
struct book b1;
```

because the data type struct book is not known to the function display().

Therefore, it becomes necessary to define the structure type struct book outside main(), so that it becomes known to all functions in the program.

* The way we can have a pointer pointing to an int, or a pointer pointing to a char, similarly we can have a pointer pointing to a struct. Such pointers are known as 'structure pointers'.

```
main()
{
    struct book
    {
        char name [25];
        char author [25];
        int callno;
    };
    struct book b1 = {"Let us C", "YPK", 101};
    struct book *ptr;

    ptr = &b1;
    pf ("\n %s %s %d", b1.name, b1.author, b1.callno);
    pf ("\n %s %s %d", ptr->name, ptr->author, ptr->callno);
}
```

* We can't use ptr.name or ptr.callno because ptr is not a structure variable but a pointer to a structure, and the dot operator requires a structure variable on its left. In such cases C provides an operator -> called an arrow operator to refer to the structure elements

b1.name	b1.price	b1.pages
'B'	180.00	550
65518	65519	65523

Array of structure:- In previou program, to store data of 100 books. we would be required to use 100 different structure variables from b1 to b100, which is definitely not very convenient. A better approach would be to use an array of structures

Eg

```
main()
{
  struct book
  {
    char name;
    float price;
    int pages;
  };
  struct book b[100];  → provides space in memory for 100 structures of the types
                         struct book.
  int i;
  for (i=0; i<=99; i++)
  {
    pf("\n Enter name, price, and pages");
    sf("%c %f %d", &b[i].name, &b[i].price, &b[i].pages);
  }
  for (i=0; i<=99; i++)
  {
    pf("\n %c %f %d", b[i].name, b[i].price, b[i].pages);
  }
```

* The syntax we use to reference each element of the array b is similar to the syntax used for arrays of ints and chars.

* b[0]'s name, price and pages in memory would be immediately followed by b[1]'s name, price and pages and so on.

Remember that on the left hand side of the '.' structure operator, there must ~~be~~ always be a structure variable, whereas on the left hand side of the '->' operator there must ~~be~~ always be a pointer to a structure.

bl.name	bl.author	bl.callno
Let Us C	YPK	101
65472	65497	65522

ptr
65472
65524

⇒ Can we not pass the address of a structure variable to a function?

Passing address of a structure variable

```
struct book
{
  char name[25];
  char author[25];
  int callno;
};
main()
{
 struct book bl = {"Let us C", "YPK", 101};
  display(&bl);
}
display(struct book *b)
{
    printf("\n %s %s %d", b->name, b->author, b->callno);
}
```

* structure struct book should be declared outside main() such that this data type is available to display() while declaring pointer to the structure

O/P Let us C YPK 101

Uses of structures:-

- → Changing the size of the cursor
- → Clearing the contents of the screen
- → Placing the cursor at an appropriate position on screen
- → Drawing any graphics shape on the screen.
- → Receiving a key from the keyboard.
- → Checking the memory size of the computer.
- → Finding out the list of equipment attached to the computer.
- → Formatting a floppy
- → Hiding a file from the directory
- → Displaying the directory of a disk
- → Sending the output to printer
- → Interacting with the mouse.

Enumerated data types → Enumerated data types gives you an opportunity to invent your own data type and define what values the variable of this data type can take. Using enumerated data types can also help you reduce programming errors.

ex

```
enum mar-status
{
    single, married, divorced, widowed
};
enum mar-status, person1, person2;
```

→ This part declares the data type and specifies its possible values. These values are called enumerators

→ This part declares variables of this data type.

Now we can give values to these variables.

```
person1 = married;
person2 = divorced;
```

* we can't use values that aren't in the original declaration.

```
person1 = unknown; → error.
```

* Internally, the compiler treats the enumerators as integers. Each values on the list of permissible values corresponds to an integer, starting with 0.

in our example single is stored as 0
married is stored as 1
divorced is stored as 2
widowed as 3

* Assigning numbers can be overridden by the programmer by initializing the enumerators to different integer values as shown below.

```
enum mar-status
{
    single = 100, married = 200, divorced = 300, widowed = 400
};
enum mar-status person1, person2;
```

explained above
⇒ Same ^ can be achieved by macros.

macros have a global scope, whereas, scope of enum can either be global (if declared outside all functions) or local (if declared inside a function).

typedef :- redefine the name of an existing variable type.

ex

```
typedef unsigned long int TWOWORDS;

TWOWORDS var1, var2;
```

instead of unsigned long int var1, var2.

* Typedef provides a short and meaningful way to call a data type. Usually uppercase letters are used to make it clear that we are ~~declaring~~ dealing with a renamed data type.

ex

```
struct employee
{
    char name[30];
    int age;
    float bs;
};
typedef struct employee Emp;
Emp e1, e2;
```

** Thus by reducing the length and apparent complexity of data types typedef can help to clarify source listing and save time and energy spent in understanding the program.

or.

```
typedef struct employee
{
    char name[30];
    int age;
    float bs;
} Emp;
Emp e1, e2;
```

```
struct employee
{
    char name[30];
    int age;
    float bs;
};
typedef struct employee *pEMP;
pEMP p;
p->age = 32;
```

Bit fields ⇒ If in a program a variable is to take only two values 1 and 0, we really need only a single bit to store it. Similarly, if a variable is to take values from 0 to 3, then two bits are sufficient to store these values. And if a variable is to take values from 0 through 7, then 3 bits will be enough and so on.

⇒ When there are several variables whose maximum values are small enough to pack into a single memory location, we can use 'bit fields' to store several values in a single integer.

FE Suppose we want to store the following data about an employee. Each employee can:

- be male or female → one bit
- be single, married, divorced or widowed. – 2 bits
- have one of the eight different hobbies. – 3 bits
- Can choose from any of the fifteen different schemes proposed by the company to pursure his/her ~~ho~~ hobby. – 4 bits

total = 10 bits.

```
struct employee
{
  unsigned gender : 1;
  unsigned mar-status : 2;
  unsigned hobby : 3;
  unsigned scheme : 4;
};
```

: → tells compiler that we are talking about ~~of~~ bit fields and the no after it tells how many bits to allot for the field.

```
#include <stdio.h>
# define MALE 0;
# define FEMALE 1;
# define SINGLE 0;
# define MARRIED 1;
# define DIVORCED 2;
# define WIDOWED 3;

Void main()
{
  struct employee
  {
     unsigned gender:1;
     unsigned mar-stat : 2;
     unsigned hobby : 3;
     unsigned scheme : 4;
  };
  struct employee e;
   e.gender = MALE;
   e.mar-stat = DIVORCED;
   e.hobby = 5;
   e.scheme = 9.
   Pf ("In Gender = %d", e.gender);
   Pf ("In Marital status = %d", e.mar-stat);
   Pf ("In Bytes occupied by e = %d", sizeof(e));
}
```

O/P Gender = 0

marital status = 2

Bytes occupied by e = 2

UNION :- Unions are derived data types. Unions and structures look like, but are engaged in totally different enterprises.

Both structures and unions are used to group a number of different variables together. But while a structure enables us to treat a number of different variables stored at different places in memory, a union enables us to treat the same space in memory as a number of different variables.

A union offers a way for a section of memory to be treated as a variable of one type on one occasion, and as a different variable of a different type on another occasion.

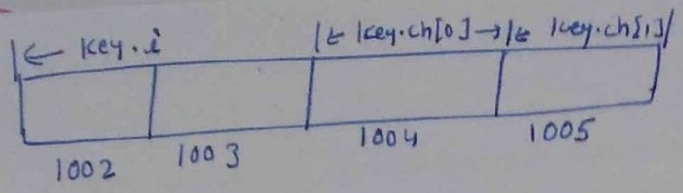
```
/* Demo of union at work */
#include <stdio.h>
{
  union a
  {
    int i;
    char ch[2];
  };
  union a key;
   key.i = 512;
   Pf("\n key.i = %d", key.i);
   Pf("\n key.ch[0] = %d", key.ch[0]);
   Pf("\n key.ch[1] = %d", key.ch[1]);
}
```

O/P

```
key.i = 512
key.ch[0] = 0
key.ch[1] = 2
```

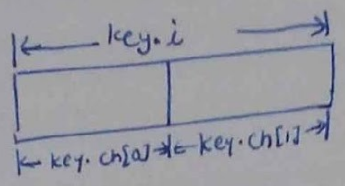
```
struct a
{
  int i;
  char ch[2];
};
struct a key;
```

→ This data type would occupy 4 bytes in memory, 2 for key.i and one each for key.ch[0] and key.ch[1]



we use a union.

```
union a
{ int i;
  char ch[2];
};
union a key;
```



* union occupies only 2 bytes in memory.

Same memory locations which are used for key.i are also being used by key.ch[0] and key.ch[1].

It means that the memory locations used by key.i can also be accessed using key.ch[0] and key.ch[1]. Now we can access the two bytes simultaneously (by using key.i) or the same two bytes individually (using key.ch[0] and key.ch[1])

<u>Union of structures</u>:- Just as one structure can be nested within another, a union too can be nested in another union. Not only that, there can be a union in a structure, or a structure in a union.

Ex

```
#include <stdio.h>
void main()
{
  struct a
  {
    int i;
    char c[2];
  };
  struct b
  { int j;
    char d[2];
  };
union z
{
  struct a key;
  struct b data;
};
union z strange;
strange.key.i = 512;
strange.data.d[0] = 0.
strange.data.d[1] = 32;
Pf("\n %d", strange.key.i);
Pf("\n %d", strange.data.i);
Pf("\n %d", strange.key.c[0]);
Pf("\n %d", strange.data
Pf("\n %d", strange.key.c[
Pf("\n %d", strange.data.d
}
```

o/p

```
512
512
0
0
32
32
```

malloc() function in C → malloc stands for memory allocation.

The malloc() function allocates single block of requested memory. It doesn't initialize memory at execution time, so it has garbage value initially. It returns NULL if memory is not sufficient.

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

FE

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n, i *ptr, sum=0;
    pf("Enter no of elements:");
    sf("%d", &n);
    ptr = (int *) malloc (n* sizeof(int)); // memory allocated using malloc
    if (ptr == NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array:");
    for (i=0; i<n; ++i)
    {
        sf("%d", ptr+i);
        sum += *(ptr+i);
    }
    printf("sum= %d", sum);
    free (ptr);
    return 0;
}
```

Dynamic memory allocation in C → The concept of dynamic memory allocation in c language enables the c programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

* Difference between static memory allocation and dynamic memory allocation.

Static memory allocation	dynamic memory allocation
→ memory is allocated at compile time	memory is allocated at run time.
→ memory can't be increased while executing program.	memory can be increased while executing program.
→ Used in array	Used in linked list.

* methods used for dynamic memory allocation.

malloc() → allocates single block of requested memory.

calloc() → allocates multiple block of requested memory.

realloc() → reallocates the memory occupied by malloc() or calloc() functions

free() → frees the dynamically allocated memory.

Calloc () function in C :- The calloc () function allocates multiple block of requested memory. It initially initialize all byte to zero. It returns NULL if memory is not sufficient.

Calloc stands for Contiguous allocation.

Syntax ptr = (cast-type*) calloc (number, byte-size)

Eg

```
# include <stdio.h>
# include <stdlib.h>
int main()
int n, i, *ptr, sum=0;
printf ("Enter no of elements");
scf ("%d", &n);
ptr = (int *) calloc (n, sizeof (int)); // memory allocated using calloc.
if (ptr==NULL)
{
  printf ("Enter element of array");
  for ( i=0; i<n; ++i)
  {
     scanf ("%d", ptr+i);
     sum += * (ptr+i);
  }
  printf ("sum= %d", sum);
  free (ptr);
return 0;
}
```

realloc() function in C:- If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. It changes the memory size.

Syntax

```
ptr = realloc (ptr, new-size)
```

free() function in C:- The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Syntax

```
free (ptr)
```

* Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. We must explicitly use free to release the space.

** An array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

** malloc function reserves a block of memory of the specified no of bytes. And it ~~rest~~ returns a pointer of void which can be casted into pointers of any form.

```
ptr = (float *) malloc (100 * sizeof (float));
```

→ allocates 400 bytes of memory. It's because the size of float is 4 bytes. And the pointer ptr holds the address of the first byte in the allocated memory. NULL pointer if memory cannot be allocated.

⇒ malloc function allocates memory and leaves the memory uninitialized, whereas the calloc function allocates memory and initializes all bits to zero.

FE

```
ptr = (float *) calloc (25, sizeof (float));
```

↳ allocates contiguous space in memory for 25 elements of type float.

Realloc FE

```
#include
# include <stdio.h>
# include <stdlib.h>
int main()
{
  int *ptr, i, n1, n2;
  Pf ("Enter size: ");
  Sf ("%d", &n1);
  ptr = (int*) malloc(n1 * sizeof (int));
  Pf ("Addresses of previously allocated memory: \n");
  for (i=0; i<n1; ++i)
    Pf ("%p \n", ptr+i);

  Pf ("\n Enter the new size: ");
  Sf ("%d", &n2);
  // reallocating the memory
  ptr = realloc (ptr, n2* sizeof (int));
  printf ("Addresses of newly allocated memory: \n");
    for (i=0; i<n2; ++i)
    Pf ("%p \n", ptr+i);
    free(ptr);
  return 0;
}
```

O/P

```
Enter size : 2
Address of previously allocated
memory
  26855472
  26855476

Enter the new size: 4
Add of newly allocated memory
  26855472
  26855476
  26855480
  26855484
```

%p → used to print the pointer type data.

– closing a file.

Ex store and use personal information of employees in comprened form.

```
struct personal
{
    unsigned sex : 1
    unsigned age : 7
    Unsigned m-status : 1
    unsigned children : 3
    unsigned          : 4
} emp;
```

File management in C :

Many real-life problems involve large volumes of data and in such situations, the console oriented I/O operations pose two major problems.

* It becomes cumbersome and time consuming to handle large volumes of data through terminals.
* The entire data is lost when either the program is terminated or the computer is turned off.

It is therefore necessary to have a more flexible approach where data can be stored on the disks and read whenever necessary, without ~~destroyed~~ ~~destori~~ destroying the data. This method employs the concept of files to store data.

A file is a place on the disk where a group of related data is stored. C supports a number of functions that have the ability to perform basic file operations, which include:

- naming a file
- opening a file
- reading data from a file
- writing data to a file
- closing a file.

There are two distinct ways to performs file operations in C.

- low-level I/O and uses UNIX system calls
- high-level I/O operation and uses functions in C's Standard I/O library.

fopen() → Creates a new file for use
↳ Opens an existing file for use.

fclose() → * Closes a file which has been opened for use.

getc() → Reads a character from a file

putc() → writes a character to a file.

fprintf() → writes a set of data values to a file

fscanf() → Reads a set of data values from a file.

getw() → Reads an integer from a file.

putw() → writes an integer to a file.

fseek() → sets the position to a desired point in the file.

~~ftell~~
ftell() → Gives the current position in the file (in terms of bytes from the start)

rewind() → Sets the position to the beginning of the file.

* There are many other functions. Not all of them are supported by all compilers. You should check your C library before using a particular I/O function.

Defining and opening a file → If we want to store data in a file in the secondary memory, we must specify certain things about the file, to the operating system. They include

1. File name
2. Data structure
3. Purpose.

1. File name:- file name is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with the extension

ex `FILE *fp;` → pointer to the data type FILE.

Structure that is defined in the I/O library

`fp = fopen("filename", "mode");` → opens the file named filename and assigns an identifier to the FILE type pointer fp.

mode can be one of the following:

r → open the file for reading only.

w → open the file for writing only.

a → open the file for appending (or adding) data to it.

** Both the filename and mode are specified as strings. They should be enclosed in double quotation marks.

When trying to open a file, one of the following things may happen:

1. When the mode is 'writing', a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
2. When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
3. If the purpose is 'reading' and if it exists, then the file is opened with the current contents safe otherwise an error occurs.

ex

```
FILE *p1, *p2;
p1 = fopen("data", "r");
p2 = fopen("results", "w");
```

The file data is opened for reading and results is opened for writing. In case the results file already exists its contents are deleted and the file is opened as a new file. If data file does not exist an error will occur.

many recent compiler include additional modes of operation. They include:

r+ → The existing file is opened to the beginning for both reading and writing

w+ → Same as w except both for reading and writing

a+ → Same as a except both for reading and writing.

Closing file:- A file must be closed as soon as all operations on it have been completed. This ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. It also prevents any accidental misuse of the file.

```
fclose( file-pointer);
```

ex

```
FILE *p1, *p2;
p1 = fopen ("INPUT", "w");
p2 = fopen ("OUTPUT", "r");
- - -
fclose (p1);
fclose (p2);
- - -
```

Input / Output operations on files:- Once a file is opened, reading out of or writing to it is accomplished using the standard I/O routines.

* getc and putc functions → The simplest file I/O functions are getc and putc. These are analogous to getchar and putchar functions and handle one character at a time.

Assume that a file is opened with mode w and file pointer fp1. Then the statement

```
putc ( c, fp1);
```

→ writes the character contained in the character variable c to the file associated with FILE pointer fp1.

getc:- Is used to read a character from a file that has been opened in read mode.

ex

```
c= getc(fp2);
```

would read a character from the file whose file pointer is fp2.

** The file pointer moves by one character position for every operation of getc or putc. The getc will return an end-of-file marker EOF, when end of the file has been reached. Therefore, the reading should be terminated when EOF is encountered.

The getw and putw Functions:- The getw and putw are integer-oriented functions. They are similar to the getc and putc functions and are used to read and write integer values. These functions would be useful when ~~the~~ we deal with only integer data.

```
putw (integer, fp);
getw (fp);
```

The fprintf and fscanf functions:- fprintf and fscanf, that can handle a group of mixed data-simultaneously. The functions fprintf and fscanf perform I/O operations that are identical to the familar printf and scanf functions, except of course that they work on files.

```
fprintf ( fp , "control string", list);
```

ex

```
fprintf( f1, "%s %d %f", name , age, 7.5);
fscanf( f2, "%s%d", item, &quantity);
```

* When end of file is reached, it returns the value EOF.

Standard library functions

stdio.h → standard input and output library is stdio.h. The functions will use the most include:

- printf() is o/p to the screen
- scanf() is read I/P from the screen
- getchar() is return characters typed on screen.
- putchar() is o/p a single characters to the screen.
- fopen() is open a file
- fclose() is close a file

Stdlib.h → is the header of the general purpose standard library of C programming language which includes functions involving memory allocation, process control, conversions and others.

Stdlib → Standard library.

members of the stdlib.h can be classified into ~~the~~ the following categories: conversion, memory, process control, sort and search, mathematics

→ atof → string to double

→ atoi → string to integer

→ atol → string to long

→ strtod → string to double

- strtol → string to long int
- strtoul → string to unsigned long int
- strtoll → string to long long int
- strtoull → string to unsigned long long int

Standard Library functions

stdio.h → standard input and output library is stdio.h. The functions will use the most include:

- printf() is o/p to the screen
- scanf() is read I/P from the screen
- getchar() is return characters typed on screen.
- putchar() is o/p a single characters to the screen.
- fopen() is open a file
- fclose() is close a file

Stdlib.h → is the header of the general purpose standard library of C programming language which includes functions involving memory allocation, process control, conversions and others.

Stdlib → Standard Library.

members of the stdlib.h can be classified into ~~the~~ the following categories: conversion, memory, process control, sort and search, mathematic

→ atof → string to double
→ atoi → string to integer
→ atol → string to long
→ strtod → string to double

- strtol → string to long int
- strtoul → string to unsigne long int
- strtoll → string to long long int
- strtoull → string to unsigned long long int

- int rand (void) → generates a pseudo-random number
- int random (void) → generates a pseudo-random number (not standard C; provided by POSIX)
- void srand (unsigned int seed) → set the rand() pseudo-random generator seed [common convention uses time() to seed]
- void srandom (unsigned int seed) → set the random() pseudo-random generator seed [common convention uses time() to seed] (not standard C; provided by POSIX)

memory allocation and deallocation →

malloc - allocate memory from the heap

calloc - allocate memory from the heap

realloc - allocate memory from the heap

free - release memory back to the heap

Process control →

/abort/ → terminate execution abnormally

atexit → register a callback function for program exit

exit → terminate program execution

getenv → retrieve an environment variable

system → execute an external command

- int rand (void) → generates a pseudo-random number
- int random (void) → generates a pseudo-random number (not standard C; provided by POSIX)
- void srand (unsigned int seed) → set the rand() pseudo-random generator seed [common convention uses time() to seed]
- void srandom (unsigned int seed) → set the random() pseudo-random generator seed [common convention uses time() to seed] (not standard C; provided by POSIX)

memory allocation and deallocation →

malloc - allocate memory from the heap

calloc - allocate memory from the heap

realloc - allocate memory from the heap

free - release memory back to the heap

Process control →

/abort/ → terminate execution abnormally

atexit → register a callback function for program exit

exit → terminate program execution

getenv → retrieve an environment variable

system → execute an external command

Sorting searching and comparison :-

→ bsearch → binary search an array

→ qsort → sort an array

mathematics :-

→ int abs (int) — absolute value of an integer

→ long int labs (long int) — absolute value of a long integer

→ div → integer division (returns quotient and remainder)

Conio.h ⇒ is a C header file used mostly by ms-Dos compiler to provide console input/output. It is not part of the C standard library as ISO C, nor it is defined by POSIX. This header declares several useful library functions for performing "console input and output" from a program. ~~most C compilers~~

The library functions declared by conio.h vary somewhat from compiler to compiler.

Kbhit → Determines if a keyboard key was pressed

cgets → Reads a string directly from the console.

cscanf → Reads formatted values directly from the console

putch → writes a character directly to the console.

cputs → writes a string directly to the console.

cprintf → Formats values and writes them directly to the console.

clrscr → clears the screen

getch → Get char entry from the console.

ctype.h :- The ctype.h header file of the C standard library declares several functions that are useful for testing and mapping characters. All the functions accepts int as a parameter, whose value must be EOF or representable as an unsigned char. All functions return non-zero (true) if the argument c satisfies the condition described and zero (~~false~~ false) if not.

→ int isalnum (int c) → This function checks whether the passed character is alphanumeric.

→ int isalpha (int c) → This function checks whether the passed character is alphabetic.

→ int iscntrl (int c) → This function checks whether the passed character is control character.

→ int isdigit (int c) → This function checks whether the passed character is decimal digit.

→ int isgraph (int c) → This function checks whether the passed character has graphical representation using locale.

→ int islower (int c) → This function checks whether the passed character is lowercase letter

→ int isprint (int c) → This function checks whether the passed character is printable.

→ int ispunct (int c) → This function checks whether the passed character is a punctuation character.

→ int ~~issp~~ isspace (int c) → This function checks whether the passed character is white space.

int isupper (int c) → This function checks whether the passed character is an uppercase letter.

int isxdigit (int c) → This function checks whether the passed character is a hexadecimal digit.

---

math.h → The math.h header defines various mathematical functions. All the functions available in this library take double as an argument and return double as the result.

→ double acos (double x) → Returns the arc cosine of x in radians.

→ double asin (double x) → Returns the arc sine of x in radians.

→ double atan (double x) → Returns the arc tangent of x in radians.

→ double atan2 (double y, double x) → Returns the arc tangent in radians of y/x based on the signs of both values to determine the correct quadrant.

→ double cos (double x) → Returns the cosine of a radian angle x.

→ double cosh (double x) → hyperbolic cosine of x.

→ double sin (double x) → sine of a radian angle x.

→ double sinh (double x) → Returns the hyperbolic sine of x.

→ double tanh (double x) → Returns the hyperbolic tangent of x.

→ double exp (double x) → Returns the value of e raised to the xth power.

String.h → string.h defines ~~one variable~~ various functions for manipulating arrays of characters.

void *memchr (const void *str, int c, size_t n) → Searches for the first occurrence of the character c (an unsigned char) in the first n bytes of the string pointed to, by the argument str.

int memcmp(const void *str1, const void *str2, size_t n) → Compares the first n bytes of str1 and str2.

void *memcpy (void *dest, const void *src, size_t n) → Copies n characters from src to dest.

void *memmove (void *dest, const void *src, size_t n) → Another function to copy n characters from str2 to str1.

void *memset (void *str, int c, size_t n) → Copies the characters of the string pointed to, by the argument str.

Char *strcat (char *dest, const char *src) → Appends the string pointed to, by src to the end of the string pointed to by dest.

char * strncat (char *dest, const char *src, size_t n) → Appends the string pointed to, by src to the end of the string pointed to, by dest up to n character long.

Process.h → header ~~file~~ file which contains function declarations and macros used in working with threads and processes.

execl, execle, execlp, execlpe → Load and execute a new child process by placing it in memory previously occupied by the parent process. Parameters are passed individually.

execv, execve, execvp, execvpe → Load and execute a new child process by placing it in memory previously occupied by the parent process. Parameters are passed as an array of pointers.

spawnl, spawnle, spawnlp, spawnlpe → Load and execute a new child process. Parameters are passed individually.

spawnv, spawnve, spawnvp, spawnvpe :- load and execute a new child process. Parameters are passed as an array of pointers

beginthread, beginthreadNT :- Create a new thread of execution within the current process.

endthread :- Terminates a thread created by beginthread.

getpaid :- returns the process identifier

Cexit :- restore interrupt vectors altered by the startup code.

UNIT-4.

Linear Search:- Linear search is a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Algo Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i+1

Step 5: Go to step 2

Step 6: Print Element x found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Ex 70, 40, 30, 11, 57, 41, 25, 14, 52

element to be searched is 41

Ex 10, 14, 19, 26, 27, 31, 33, 35, 42, 44

element to be searched is 33

0 1 2 3 4 5 6

Binary Search:- Binary Search is a fast search algorithm with run-time Complexity of $O(\log n)$. This search algorithm works on the principle of divide and conquer. For this algorithm to work properly the data collection should be in the sorted ~~form~~.

Ex 10, 14, 19, 26, 27, 31, 33, 35, 42, 44

Algorithm: Binary-Search (a, lower-bound, upper-bound, val)

Step1: set beg = lower-bound, end = upper-bound, pos = -1

Step2: repeat step 3 and 4 while beg $\leq$ end.

Step3: set mid = (beg + end)/2

Step 4: if a[mid] = val

set pos = mid

print pos

go to step 6

else if a[mid] > val

set end = mid-1

else

set beg = mid+1

end if

end of loop

Step5: if pos = -1

print "value is not present in the array"

end of if

Step 6: exit

Q

0	1	2	3	4	5	6	7	8	9
10,	14,	19,	26,	27,	31,	33,	35,	42,	44

item = 31

mid = low + (high − low)/2

$0 + (9-0)/2 = 4.5 \Rightarrow 4$ is the mid of the array

10, 14, 19, 26, 27, 31, 33, 35, 42, 44.

Now, we compare the value stored at location 4, with value being searched i.e. 31.

31, 33, 35, 42, 44

low = mid + 1

mid = low + (high − low)/2

~~0 + (4 − 0)/2~~ $5 + (9-5)/2$

~~= 2 ⇒ 35~~ $\Rightarrow 5+2$

~~गलतीकिया~~ $= 7$

The value stored at location 7 is not match, rather it is more than what we are looking for, so the value must be in the lower part from this location.

Hence, we calculate the mid again, This time it is 5.

We compare the value stored at location 5 with our target value.

We conclude that the target value 31 is stored at location 5

Bubble sort algorithm:- Bubble sort is a simple algorithm. This sorting algo is comparison-based algorithm in which each pair of adjacent element is compared and the elements are swapped if they are not in order. This algo is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the no of items.

Algorithm

```
begin BubbleSort (list)
    for all elements of list
        if list [i] > list [i+1]
            swap (list [i], list [i+1])
        end if
    end for
    return list
end BubbleSort
```

ex 14, 33, 27, 35, 10

14, 33, 27, 35, 10

14, 33, 27, 35, 10

14, 27, 33, 35, 10

14 27 33 35 10

14 27 33 10 35

14, 27, 10, 33, 35

14, 10, 27, 33, 35

10, 14, 27, 33, 35

INSERTION ALGORITHM → An element which is to be 'inserted' in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where n is the no of items.

Algo

Step 1: If it is the first element, it is already sorted. return 1;

Step 2: Pick next element

Step 3: Compare with all elements in the sorted sub-list.

Step 4: Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5: Insert the value

Step 6: Repeat until list is sorted

ex 14, 33, 27, 10, 35, 19, 42, 44

1. 14 33 27 10 35 19 42 44
2. 14 27 33 10 35 19 42 44
3. 14 27 10 33 35 19 42 44
4. 14 10 27 33 35 19 42 44
5. 10 14 27 33 35 19 42 44
6. 10 14 27 33 19 35 42 44
7. 10 14 27 19 33 35 42 44
8. 10 14 19 27 33 35 42 44
9. 10 14 19 27 33 35 42 44

Selection sort algorithm:- Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes a part of the sorted array.

This algorithm is not suitable for large dat sets as its average and worst case complexities are of $O(n^2)$, where n is the no of items.

~~Ex 14, 33~~

Algo

Step 1: Set MIN to location 0

Step 2: Search the minimum element in the list

Step 3: Swap with value at location MIN

Step 4: Increment MIN to point to next element

Step 5: Repeat until list is sorted.

ex 14, 33, 27, 10, 35, 19, 42, 44

1) 10 33 27 14 35 19 42 44

2) 10 14 27 33 35 19 42 44

10 14 19 33 35 27 42 44

10 14 19 27 35 33 42 44

10 14 19 27 33 35 42 44