UNIT 6

Pointers

What is a pointer in C?

Pointer is a variable that stores/points the address of another variable. Pointer variable can only contain the address of a variable of the same data type.

eg.

```
int *ptr;   //pointer declaration
int a=5;
ptr=&a;   //address of variable a is assigning to pointer variable ptr
```

Here ptr is a pointer variable that only contains the address of integer data type.

What are the advantages of using pointer?

1) Pointers are more efficient in handling arrays.
2) Pointers can be used to return multiple values from functions via function arguments.
3) The pointer enables us to access a variable that is defined outside function.
4) Pointer allows c to support dynamic memory management.
5) Pointer reduce length and complexity of programs.
6) They increase the execution speed and thus reduce the program execution time.
7) Pointers provide efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stack and trees.

Pointer declaration

Pointer variable is declared with an asterisk (*) operator before a variable name. This operator is called pointer/indirection or dereference operator.

Syntax: data_type *pointer_variable_name;

Data type of the pointer must be same as the data type of the variable to which the pointer variable is pointing.

eg. int *ptr;

Declares the variable ptr which is a pointer variable that points to an integer data type.

Similarly the statement

float *x;

Declares the variable x which is a pointer variable that points to a float data type.

Initialization of pointers

Pointer initialization is the process of assigning address of variable to a pointer variable. In C programming language ampersand (&) operator is used to determine the address of variable. The & (immediately preceding a variable name) returns the address of variable associated with it.

Pointer variable always points to variables of same data type.

```
int main()
{
    int a=10;
    int *ptr;      //pointer declaration
    ptr=&a;        //pointer initialization
}
```

Here, pointer variable ptr of integer type is pointing the address of integer variable a. So this is valid.

Let's have another example

```
int main()
{
   float a;
   int *ptr;
   ptr=&a;      //ERROR, type mismatch
}
```

Here, pointer variable of integer type ptr is pointing the address of float variable a. So this is invalid.

Once, address of variable is assigned to pointer, then to access the value of variable, pointer is dereferenced using indirection/dereference operator (*).

Example:

```
#include<stdio.h>
int main ()
{
  int a=5;
  int *ptr;                  //declaration of pointer variable
  ptr=&a;                    //initializing the pointer
  printf("%d\n",a);          //prints the value of a
  printf("%d\n",*ptr);       //prints the value of a
  printf("%d\n",*(&a));      //prints the value of a
  printf("%p\n",&a);         //prints the address of a
  printf("%p\n",ptr);        //prints the address of a
  printf("%p",&ptr);         //prints the address of pointer variable ptr
  return 0;
}
```

Pointer Operators

A pointer is a variable that stores the memory address of another variable. Instead of holding a direct value, it holds the address where the value is stored in memory. There are 2 important operators that we will use in pointers concepts i.e.

- ✓ Dereferencing operator(*) used to declare pointer variable and access the value stored in the address.
- ✓ Address operator(&) used to returns the address of a variable or to access the address of a variable to a pointer.

```
#include <stdio.h>
int main()
{
   int m ;
   m=5;
   int *ptr;
   ptr = &m;
   printf("The Memory Address of Variable m is: %p\n", &m);
   printf("The Memory Address of Variable m is using ptr: %p\n", ptr);
   printf(" value of m=%d\n",*(&m));
   printf(" value of m=%d",*(ptr));
   return 0;
}
```

Bad pointer

- ➢ When a pointer variable is declared and if any valid address is not assigned to it then pointer is known as bad pointer. A dereference operation on a bad pointer is a serious runtime error.
- ➢ Each pointer must be assigned a valid address before it can support dereference operations. Before that pointer is bad and must not be used.
- ➢ It is always a best practice to initialize a pointer NULL values, when they are declared and check for whether the pointer is NULL pointer when using the pointer.

Void pointer

Write a short note on: Void pointer

Void pointer is a special type of pointer that can point any data type (int or float or char or double). Using void pointer, the pointed data cannot be dereferenced (cannot accessed the value at the address stored in pointer variable).so that reason we will always have to change type of void pointer to some other pointer type that points to a concrete data type before dereferencing it This is done by performing type-casting.

Declaration: void *pointer_name;

Example:

```
#include<stdio.h>
int main()
{
  int a=10;
  float b=4.5;
  void *ptr;
  ptr=&a;
  printf("a=%d",*(int*)ptr);
  ptr=&b;
  printf("b=%f",*(float*)ptr);
  return 0;
}
```

Output:

```
a=10
b=4.500000
```

Null pointer

- A Null pointer is a pointer which is pointing to nothing.
- Pointer which is initialized with NULL value is considered as NULL pointer.
- We can define a null pointer using a predefined constant NULL, which is defined in header files stdio.h , stdef.h, stdlib.h.
- Some of the most common use cases for NULL are
  a) To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.
     int * ptr = NULL;
  b) To check for null pointer before accessing any pointer variable. By doing so, we can perform error handling in pointer related code e.g. dereference pointer variable only if it's not NULL.
     if(ptr != NULL) /*We could use if (ptr) as well*/
     { /*Some code*/}
     else
     { /*Some code*/}

Pointer Arithmetic

As a pointer holds the memory address of variable, some arithmetic operations can beperformed with pointers. They are as follows.

- ✓ Addition of an integer to a pointer and increment operation
- ✓ Subtraction of an integer to a pointer and decrement operation
- ✓ Subtraction of a pointer from another pointer of same type.

If arithmetic operations done values will be incremented or decremented as per data type chosen.

Let ptr=&arr[0]=1000 i.e Base address of array

int type pointer (2-byte space)	**float type pointer (4-byte space)**	**char type pointer (1-byte pointer)**
ptr++=ptr+1=1000+2= 1002 is the address of next element.	ptr++=ptr+1=1000+4= 1004 is the address of next element	ptr++=ptr+1=1000+1= 1001 is the address of next element
ptr=ptr+4=1000+(2*4)=1008 ie. Address of 5^{th} integer type element (&arr[4])	ptr=ptr+4=1000+(4*4)=1016 ie. Address of 5^{th} float type element (&arr[4])	ptr=ptr+4=1000+(1*4)=1004 ie. Address of 5^{th} char type element(&arr[4])

Note: Similar operation can be done for decrement.

Example for pointer increment/decrement

```
#include<stdio.h>
int main()
{
   int arr[5] = {10,20,30,40,50};
   int *ptr1,*ptr2;
   ptr1=&arr[0];
   ptr1++;
   printf("value %d has address %p\n",*ptr1,ptr1);    //points to 2nd element
   ptr2=&arr[4];
   ptr2-- ;
   printf("value %d has address %p\n",*ptr2,ptr2);    //points to 4th element
   return 0;
}
```

Example for pointer addition/subtraction with integer constant

```
#include<stdio.h>
int main()
{
   int arr[5] = {10,20,30,40,50};
   int *ptr1,*ptr2;
   ptr1=&arr[0];
   ptr1=ptr1+4;
   printf("value %d has address %p\n",*ptr1,ptr1);   //points to 5th element
   ptr2=&arr[4];
   ptr2=ptr2-4 ;
   printf("value %d has address %p\n",*ptr2,ptr2);    //points to 1st element
   return 0;
}
```

One pointer variable can be subtracted from another provided that both variables point to the element of same array.

```
#include<stdio.h>
int main()
{
int arr[5]={1,2,3,4,5};
int *ptr1,*ptr2;
ptr1=arr;
ptr2=&arr[4];
printf("Difference of two pointer =%d",ptr2-ptr1);
return 0;
}
```

Some other Operations that can be performed on pointer

1) **A pointer variable can be assigned the assigned the address of an ordinary variable.**

```
#include<stdio.h>
int main()
{
int a=5;
int *ptr;
ptr=&a;
printf("value of a=%d\n",a);
printf("Address of a=%p",ptr);
return 0;
}
```

2) **Content of one pointer variable can be assigned to other pointer variable provided they point to same data type.**

```
#include<stdio.h>
 int main()
{
 int arr[5]={1,2,3,4,5};
 int *ptr1,*ptr2;
 ptr1=&arr[0];
 ptr2=ptr1; //assign element of ptr1 to ptr2
 printf("ptr1 contains %p and ptr2 contains %p",ptr1,ptr2);
 return 0;
}
```

3) **Two pointers' variables can be compared provided both pointers point to object of same data type.**

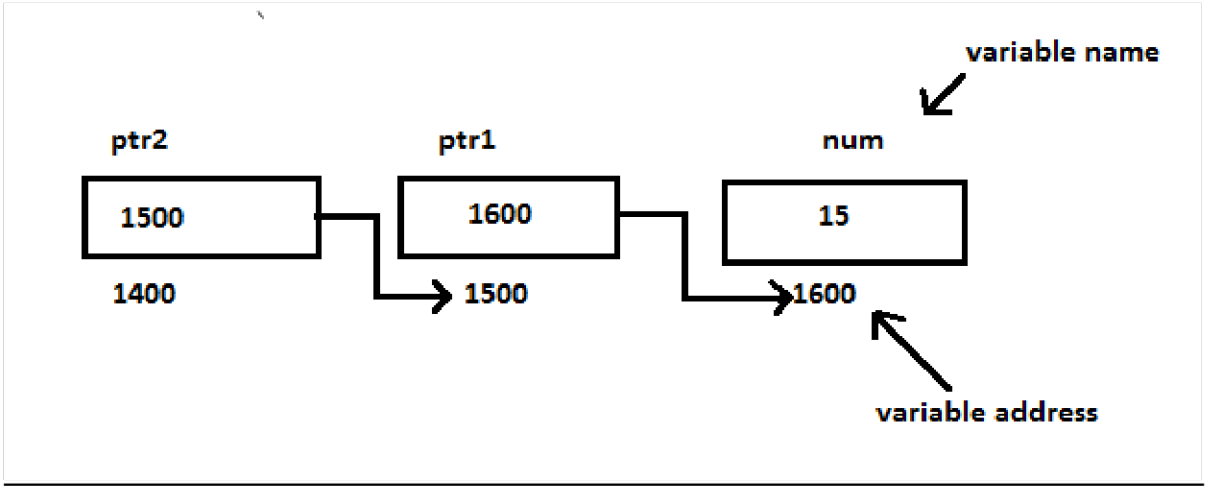
```
#include<stdio.h>
int main()
{
int arr[5]={1,2,3,4,5};
int *ptr1,*ptr2;
ptr1=&arr[0];
ptr2=&arr[4];
if(ptr2>ptr1)
{
printf("ptr2 is far from ptr1");
}
else
{
printf("ptr1 is far from ptr2");
}
return 0;
}
```

Invalid pointer operations

- Addition of two pointer (ptr1+ptr2)
- Multiplication of two pointer(ptr1*ptr2)
- Addition or subtraction of float or double data type to or from a pointer
- Multiplication of pointer with constant (ptr1*5)
- Division of two pointer or with constant (ptr1/5)

Double indirection (Double pointer/chain of pointer /pointer to pointer)

When pointer holds the address of another pointer then such type of pointer is called double pointer/chain of pointer.



As per diagram, here ptr1 is a normal pointer that holds the address of an integer variable num. There is another pointer ptr2 in the diagram that holds the address of another pointer ptr1, the pointer ptr2 here is pointer to pointer (or double pointer)

A variable that is pointer to pointer must declared using two indirection operator symbol in front of name.eg. int **ptr2;

Example

```
#include<stdio.h>
int main()
{
  int num=15;
  int *ptr1;
  int **ptr2;
  ptr1=&num;
  ptr2=&ptr1;
  printf("num=%d\n",*ptr1);
  printf("num=%d",**ptr2);
  return 0;
}
```

Output

```
num=  15
num=  15
```

- ✓ In above program we can observe that the statement printf("num=%d",**ptr2); gives the value of num=15.
- ✓ So, to access the value indirectly pointed by a pointer to pointer, we can use double indirection operator.

Passing pointer to function

Pointers can also be passed to a function as an argument. When we pass a pointer as an argument then the address of the variable is passed instead of the value. So any change made by the function using the pointer is reflected in actual argument that is passed through function. This mechanism of passing argument to function is known as call by reference.

Program to illustrate passing pointer to function

```
#include<stdio.h>
void addGraceMarks(int *m);
int main()
{
        int marks;
        printf("Enter the actual marks\n");
        scanf("%d",&marks);
        addGraceMarks(&marks);
        printf("The final marks is:%d",marks);
        return 0;
}
```

```
void addGraceMarks(int *m)
{
        *m=*m+10;
}
```

Write a Program to find sum of three numbers by passing pointer to function.

```
#include<stdio.h>
int sum(int *x,int *y,int *z);
int main()
{
int a,b,c,result;
printf("Enter any three numbers\n");
scanf("%d%d%d",&a,&b,&c);
result=sum(&a,&b,&c);
printf("Sum of three numbers=%d",result);
return 0;
}
int sum(int *x,int *y,int *z)
{
int r;
r=*x+*y+*z;
return r;
}
```

Write a Program to convert uppercase letter into lower and vice versa passing pointer to function.

```
#include<stdio.h>
void conversion(char *);
int main()
{
        char ch;
        printf("Enter the character\n");
        scanf("%c",&ch);
        conversion(&ch);
        printf("The corresponding character is:%c",ch);
        return 0;
}
void conversion(char *c)
{
  if(*c>='a'&&*c<='z')
  {
         *c=*c-32;
  }
  else if (*c>='A'&&*c<='Z')
  {
         *c=*c+32;
  }
}
```

What are the advantages of using pointer in a function?

The advantages of using pointers in functions are:

- ✓ Address of variable can be passes to the function so that changes to value of arguments in calling function can be easily reflected in called function.
- ✓ In general function cannot return more than one values, but it can be possible using concept of pointers.
- ✓ Arrays and structures can be passes to the function efficiently.
- ✓ It is possible to pass a portion of an array rather than an entire array to a function using pointer.
- ✓ The use of pointer as a function argument permits the corresponding data items to be altered globally from within the function.

Write a program with user defined function using pointer to convert all the upper case to lower case and vice-versa in string given by the user.

```
#include<stdio.h>
void conversion(char *s);
int main( )
{
        char str[50];
        printf("Enter the string\n");
        gets(str);
        conversion(str);
        printf("String after conversion");
        puts(str);
        return 0;
}
void conversion(char *s)
{
   int i;
   for(i=0; *(s+i) != '\0';i++)
   {
        if(*(s+i)>='a'&&*(s+i)<='z')
        {
          *(s+i)=*(s+i)-32;
        }
        else if(*(s+i)>='A'&&*(s+i)<='Z')
         {
          *(s+i)=*(s+i)+32;
          }
      }
 }
```

Returning multiple values from function

- Does function return single or multiple value? When and how a function will return single or multiple value? Illustrate with examples.
- How can function return multiple values? Explain with example?

Normally, (when function arguments are passed by value) more than one value cannot be returned at a time using function. But when we use pointer in function, we can return more than one value.

When we pass the function argument with their address and make changes in their value using pointer, then any change made by the function is reflected in actual argument that is passed through function. Hence, we can pass any number of parameters as reference, which we want to return from function.

Example:

```
#include<stdio.h>
void areaperi(int r,float *area,float *peri);
int main()
{
  int rad;
  float a,p;
  printf("Enter the radious\n");
  scanf("%d",&rad);
  areaperi(rad,&a,&p);
  printf("Area of circle=%f",a);
  printf("Perimeter of circle=%f",p);
  return 0;
}
void areaperi(int r,float *area,float *peri)
{
        *area=3.14*r*r;
        *peri=2*3.14*r;
}
```

Pointer and arrays

Pointers and 1D-Array

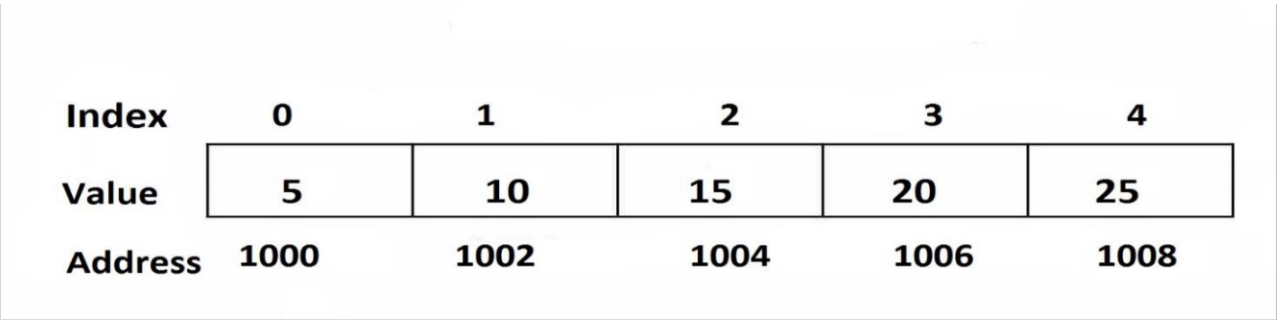
When the array is declared,

- Compiler allocates the sufficient amount of memory to contain all the elements of array in contiguous memory location.
- Base address ie. address of the first element of the array is allocated by the compiler.

Suppose, we declare an array arr.

int arr[5]={5,10,15,20,25};

Assuming that base address of arr is 1000 and each integer requires two bytes, the five elements will be stored as follows.



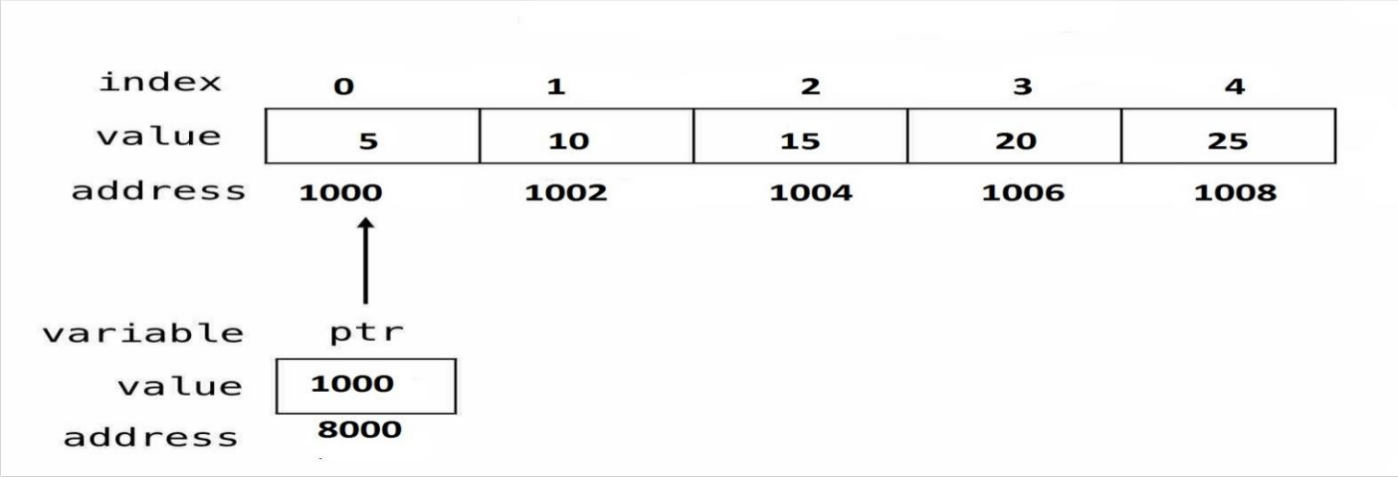
Here variable **arr** holds the address of the first element of the array.ie. Points at the starting memory of address.

So, arr contains the address of arr[0] ie.1000

If we want to declare ptr as integer pointer then we can make the pointer ptr to point the array arr by following assignment.

int *ptr;

ptr=arr; **OR** ptr=&arr[0];



Now we can access every element of array arr by incrementing the value of ptr to move one element to another.

Illustration:

int arr[5]={5,10,15,20,25};

int *ptr, i;

ptr=arr; //similar to ptr=&arr[0]

By assuming array starts at location 1000

&arr[0]=1000	arr[0]=5	ptr+0=1000	*(ptr+0)=5
&arr[1]=1002	arr[1]=10	ptr+1=1002	*(ptr+1)=10
&arr[2]=1004	arr[2]=15	ptr+2=1004	*(ptr+2)=15
&arr[3]=1006	arr[3]=20	ptr+3=1006	*(ptr+3)=20
&arr[4]=1008	arr[4]=25	ptr+4=1008	*(ptr+4)=25

Thus

Accessing value	Accessing Address

printf("%d",*(ptr+i)); //displays the arr[i] value
printf("%d",*ptr); //display the arr[0] value
printf("%d",*(arr+i); //displays the arr[i] value
	printf("%p",(ptr+i)); //displays the address of arr[i]
	printf("%p", ptr); //display the address of arr[0]
	printf("%p",(arr+i)); //displays the address of arr[i]

Program to print the value and address of elements of array.

```
#include<stdio.h>
int main()
{
int *ptr,i;
int arr[5]={5,10,15,20,25};
ptr=arr;
for(i=0;i<5;i++)
{
printf("value of arr[%d]=%d\n",i,*(ptr+i));
printf("address of arr[%d]=%p\n",i,ptr+i);
}
return 0;
}
```

Write a program to input n number in array and display it using pointer.

```
#include<stdio.h>
int main()
{
int arr[100],i,n,*ptr;
ptr=arr;
printf("Enter the number of elements you want to enter\n");
scanf("%d",&n);
printf("Enter %d elements\n",n);
for(i=0;i<n;i++)
{
scanf("%d",(ptr+i));
}
printf("Entered elements are\n");
for(i=0;i<n;i++)
{
printf("%d\n",*(ptr+i));
}
return 0;
}
```

Assignment:

Write a Program to input 5 elements in an array and display it using pointer.

Write a Program using pointer to read an array of integers. Next add the elements in the array and display sum on screen.

```
#include<stdio.h>
int main()
{
  int arr[100],n,i,*ptr,sum=0;
  ptr=arr;
  printf("Enter number of  elements you want to enter\n");
  scanf("%d",&n);
  printf("Enter %d elements\n",n)      ;
  for(i=0;i<n;i++)
  {
    scanf("%d",(ptr+i));
   }
  for(i=0;i<n;i++)
  {
  sum=sum+*(ptr+i);
  }
  printf("The sum of all elements is %d",sum);
  return 0;
}
```

Write a program to sort n integer values in an array using pointer.

```
#include<stdio.h>
int main()
{
  int arr[100],n,i,j,temp,*ptr;
  ptr=arr;
  printf("Enter number of  elements you want to enter\n");
  scanf("%d",&n);
  printf("Enter %d elements\n",n)      ;
  for(i=0;i<n;i++)
  {
    scanf("%d",(ptr+i));
  }
  for(i=0;i<n-1;i++)
  {
          for(j=0;j<n-i-1;j++)
          {
             if(*(ptr+j)>*((ptr+j)+1))
             {
                    temp=*(ptr+j);
                    *(ptr+j)=*((ptr+j)+1);
                    *((ptr+j)+1)=temp;
             }
          }
  }
```

```
 printf("The sorted array are\n");
 for(i=0;i<n;i++)
 {
        printf("%d\n",*(ptr+i));
 }
 return 0;
}
```

Write a program using pointers to read in an array of integers and prints its elements in reverse order.

```
#include<stdio.h>
int main()
{
 int arr[100],i,n,*ptr;
 ptr=arr;
 printf("Enter number of  elements you want to enter\n");
 scanf("%d",&n);
 printf("Enter %d elements\n",n)      ;
 for(i=0;i<n;i++)
 {
        scanf("%d",(ptr+i));
 }
 printf("The array elements are\n");
 for(i=0;i<n;i++)
 {
        printf("%d\n",*(ptr+i));
 }
  printf("The array elements in reverse order are\n");
  for(i=n-1;i>=0;i--)
  {
        printf("%d\n",*(ptr+i));
  }
 return 0;
}
```

Write a program to input n number in an array and find highest and lowest element using pointer.

```
#include<stdio.h>
int main()
{
 int arr[100],n,i,*ptr,high,low;
 ptr=arr;
 printf("Enter number of  elements you want to enter\n");
 scanf("%d",&n);
 printf("Enter %d elements\n",n);
```

```
 for(i=0;i<n;i++)
 {
  scanf("%d",(ptr+i));
 }
 high=*ptr;
 low= *ptr;
 for(i=0;i<n;i++)
 {
   if(*(ptr+i)>high)
   {
     high=*(ptr+i);
    }
     if(*(ptr+i)<low)
     {
      low=*(ptr+i);
     }
 }
 printf("Highest element=%d\n",high);
 printf("Lowest element=%d\n",low);
 return 0;
}
```

Write a Program to input n number in an array and find sum of odd and even elements and count them using pointer.

```
#include<stdio.h>
int main()
{
 int arr[100],n,i,*ptr,esum=0,ecount=0,osum=0,ocount=0;
 ptr=arr;
 printf("Enter number of  elements you want to enter\n");
 scanf("%d",&n);
 printf("Enter %d elements\n",n)      ;
 for(i=0;i<n;i++)
  {
           scanf("%d",(ptr+i));
  }
 for(i=0;i<n;i++)
 {
         if(*(ptr+i)%2==0)
         {
                  esum=esum+*(ptr+i);
                  ecount++;
         }
         else
         {
          osum=osum+*(ptr+i);
          ocount++;
         }
 }
```

```
  printf("Sum of even elements=%d\n",esum);
  printf("Number of even elements=%d\n",ecount);
  printf("Sum of odd elements=%d\n",osum);
  printf("Number of odd elements=%d\n",ocount);
  return 0;
}
```

Write a Program to input n number in an array and check the given number is present or not using pointer

```
#include<stdio.h>
int main()
{
  int arr[100],n,i,*ptr,num,flag=0;
  ptr=arr;
  printf("Enter number of  elements you want to enter\n");
  scanf("%d",&n);
  printf("Enter %d elements\n",n)      ;
  for(i=0;i<n;i++)
  {
   scanf("%d",(ptr+i));
  }
 printf("Enter a number you want to search\n");
 scanf("%d",&num);
 for(i=0;i<n;i++)
  {
          if(*(ptr+i)==num)
          {
            flag=1;
            break;
          }
  }
  if(flag==1)
  {
          printf("Your number is found\n");
  }
  else
  {
          printf("Your number is not found\n");
  }
  return 0;
}
```

Pointers and two-dimensional Arrays

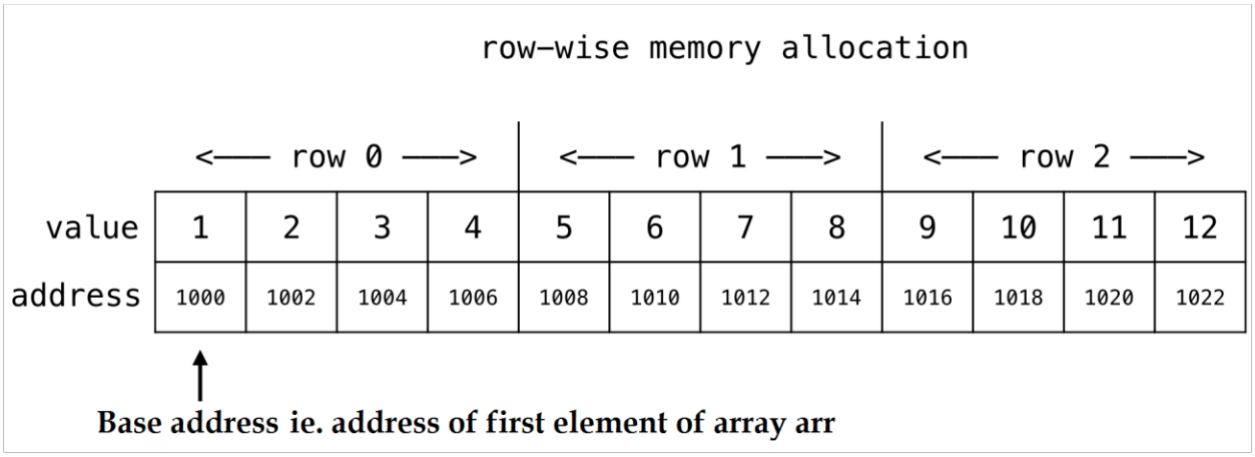
In a two-dimensional array, we can access each element by using two subscripts, where first subscript represents the row number and second subscript represents the column number. The elements of 2-D array can be accessed with the help of pointer notation also.
Suppose arr is a 2-D array, we can access any element arr[i][j] of the array using the pointer expression *(*(arr + i) + j).

Let us take a two dimensional array arr[3][4]:

int arr[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };

	Col 1	Col 2	Col 3	Col 4
Row 1	1	2	3	4
Row 2	5	6	7	8
Row 3	9	10	11	12

Since memory in computer is organized linearly it is not possible to store the 2-D array in rows and columns. The concept of rows and columns is only theoretical, actually a 2-D array is stored in row major order i.e rows are placed next to each other. The following figure shows how the above 2-D array will be stored in memory.



- ✓ Each row of the 2-D array treated as a 1-D array, so a two-dimensional array can be considered as a collection of one-dimensional arrays that are placed one after another.
- ✓ So here arr is an array is an array of 3 elements where each element is a 1-D array of 4 integers.
- ✓ We know that the name of an array is a constant pointer that points to 0^{th} 1-D array and contains address 1000. Since arr is a 'pointer to an array of 4 integers', according to pointer arithmetic the expression arr + 1 will represent the address 1008 and expression arr + 2 will represent address 1016.
- ✓ So, we can say that arr points to the 0^{th} 1-D array, arr + 1 points to the 1^{st} 1-D array and arr + 2 points to the 2^{nd} 1-D array.

arr	points to 0th element of arr	points to 0th 1D array	1000
arr+1	points to 1th element of arr	points to 1th 1D array	1008
arr+2	points to 2th element of arr	points to 2th 1D array	1016

- ✓ arr + i Points to ith element of arr →Points to ith 1-D array
- ✓ Since arr + i points to ith element of arr, on dereferencing it will get ith element of arr which is of course a 1-D array. Thus, the expression *(arr + i) gives us the base address of ith 1-D array.
- ✓ Similarly, *(arr + i) + j will represent the address of jth element of ith 1-D array. On dereferencing this expression we can get the jth element of the ith 1-D array.
- ✓ For example *(arr + 1) + 1 will represent the address of 1st element of 1st 1-D array and *(arr+2)+2 will represent the address of 2nd element of 2nd 1-D array.
- ✓ On dereferencing expression *(arr + 1) + 1 i.e *(*(arr + 1)+1) will give arr[1][1] element of an array.
- ✓ So, in 2-D array, we can access any element arr[i][j] of the array using the pointer expression *(*(arr + i) + j)

Example:

```
#include<stdio.h>
int main()
{
int arr[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };
int i, j;
printf("Given 2D array elements are\n");
for(i = 0; i < 3; i++)
{
  for(j = 0; j < 4; j++)
  {
    printf("arr[%d][%d]=%d\t", i, j, *( *(arr + i) + j) );
  }
  printf("\n");
}
return 0;
}
```

Write a Program to add two matrix of order m*n by using the concept of pointer.

```
#include<stdio.h>
int main()
{
int a[20][20],b[20][20],c[20][20],i,j,m,n;
printf("Enter the order of matrix\n");
scanf("%d%d",&m,&n);
printf("Enter the first matrix\n");
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
{
  scanf("%d",(*(a+i)+j));
}
}
```

```
printf("Enter the second matrix\n");
for(i=0;i<m;i++)
{
        for(j=0;j<n;j++)
        {
        scanf("%d",(*(b+i)+j));
        }
}
for(i=0;i<m;i++)
{
        for(j=0;j<n;j++)
        {
        *(*(c+i)+j)=*(*(a+i)+j)+ *(*(b+i)+j);
        }
}
printf("Addition of two Matrix\n");
for(i=0;i<m;i++)
{
        for(j=0;j<n;j++)
        {
        printf("%d\t",*(*(c+i)+j));
        }
        printf("\n");
}
return 0;
}
```

Write a program to input m*n matrix and find the sum of all elements.

```
#include<stdio.h>
int main()
{
int arr[20][20],i,j,m,n,sum=0;
printf("Enter the order of matrix\n");
scanf("%d%d",&m,&n);
printf("Enter %d elements of matrix\n",m*n);
for(i=0;i<m;i++)
{
  for(j=0;j<n;j++)
  {
    scanf("%d",(*(arr+i)+j));
  }
}
for(i=0;i<m;i++)
{
  for(j=0;j<n;j++)
    {
     sum=sum+*(*(arr+i)+j);
    }
}
```

```
printf("sum of matrix is %d",sum);
return 0;
}
```

Array of pointers

- ✓ Array of pointers is a collection of address.
- ✓ The address present in the array of pointers can be address of variable or address of array element.

Array of pointers can be declared as:

datatype *pointer_name[size];

eg.int *ptr[4];

This statement declares an array of 4 pointers, each of which points to an integer value. The first pointer is called ptr[0] ,the second is ptr[1] ,third is ptr[2] and fourth is ptr[3].

Example:

```
#include<stdio.h>
int main()
{
   int a=5,b=6,c=7,d=8,i;
   int *ptr[4];
   ptr[0]=&a;
   ptr[1]=&b;
   ptr[2]=&c;
   ptr[3]=&d;
   for(i=0;i<4;i++)
   {
     printf("%d",*ptr[i]);
   }
    return 0;
}
```

Difference between array of pointers and pointer to arrays

Array of pointers	Pointers to Array
1. An array of pointers is a list of variables that have addresses (memory locations).	1. A pointer to an array is a variable that directs you to a memory location that contains a list.
2. **Syntax**: data_type *variable_name[size];	2. **Syntax**: data_type (*variable_name)[size];
3. **Example:** int *ptr[4] ; must be read as *'a is an array of 4 pointers to int'*	**3. Example:** int (*ptr)[4]; Here, ptr is a pointer to an array of 4 integers.

4. **Program**
 	
#include<stdio.h>
int main()
{
int a=5,b=6,c=7,d=8,i;
int *ptr[4];
ptr[0]=&a;
ptr[1]=&b;
ptr[2]=&c;
ptr[3]=&d;
for(i=0;i<4;i++)
{
printf("%d\n",*ptr[i]);
}
return 0;
}
	4. **Program**
	#include<stdio.h>
	int main()
	{
	int arr[3][4] ={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
	int i, j;
	int (*ptr)[4];
	/*ptr is a pointer to an array of four integers */
	ptr = arr;
	for(i = 0; i < 3; i++)
	{
	for(j = 0; j < 4; j++)
	{
	printf("arr[%d][%d]=%d\t", i, j, *(*(ptr + i) + j));
	}
	printf("\n");
	}
	return 0;
	}

Difference between array and pointer

Array	Pointer
1. Array is a collection of similar data items under a common variable name.	1. Pointer is a variable that points/stores the address of another variable.
2. An array name represents a fixed memory address that cannot be changed. We can index it, but we can't change the address it refers to.	2. As pointer is a variable contains address, we can change the address stored in that variable to point to something else.
3. *Declaration:* ***data_type array_name[size];*** eg.int arr[10];	3. *Declaration*: **data_type *pointer_variable_name;** eg. int *ptr;
4. We cannot use name of array as variable. eg. int arr[10]; arr=ptr; // is not allowed	4. In pointer variable we can assign the address of another variable. eg. int *ptr; ptr=arr; // is allowed
5. An array can store the number of elements, mentioned in the size of array variable	5. A pointer variable can store the address of only one variable at a time.

Pointers and strings

As the string is an array of characters, the name of string variable is pointer to the first character of the string and can be used to access and manipulate the characters.

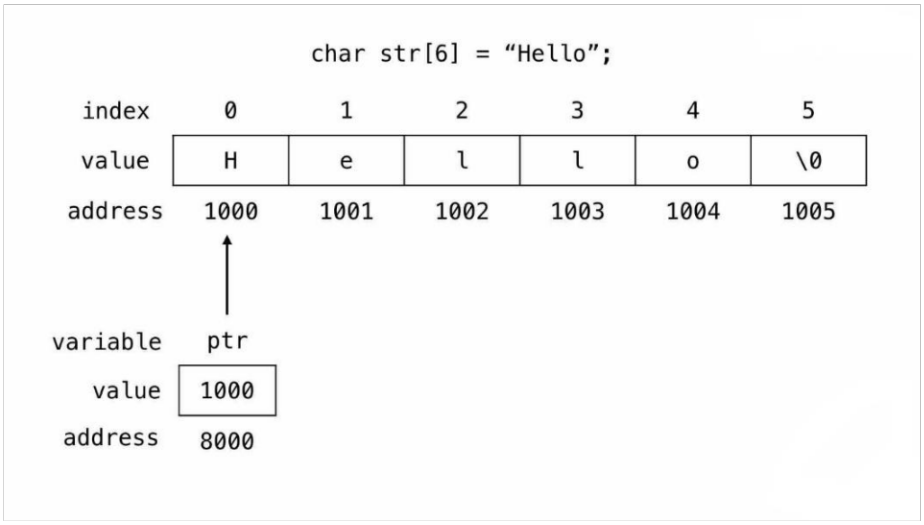
eg.

char str[6] = "Hello";

char *ptr = str; // assigning the address of the string str to the pointer ptr.

Here, the variable name of the string str holds the address of the first element of the array i.e., it points at the starting memory address.

We can represent the character pointer variable ptr as follows.



The pointer variable ptr is allocated memory address 8000 and it holds the base address of the string variable str i.e., 1000.

Accessing string via pointer

To access and print the elements of the string we can use a loop and check for the '\0 ' null character. In the following example we are using while loop to print the characters of the string variable str.

```
#include <stdio.h>
int main()
{
        char str[6] = "Hello";   // string variable
        char *ptr = str;         // pointer variable
        while(*ptr != '\0')
        {
         printf("%c", *ptr);     // print the string
          ptr++;                 // move the ptr pointer to the next memory location
        }
        return 0;
}
```

Passing array element to a function using pointer

Write a program to pass array element to a function using pointer.

```
#include<stdio.h>
void display(int *p,int m);
int main()
{
int arr[100];
int *ptr,n,i;
ptr=arr;
printf("Enter the number of elments\n");
scanf("%d",&n);
printf("Enter %d array elements\n",n);
for(i=0;i<n;i++)
{
        scanf("%d",(ptr+i));
}
printf("The array elements are\n");
display(arr,n);
return 0;
}
void display(int *p,int m)
{
        int i;
        for(i=0;i<m;i++)
        {
                printf("%d\n",*(p+i));
        }
}
```

Write a program to pass the elements of an array to a function using pointer and use the function to find the sum of the elements and print the sum.

```
#include<stdio.h>
void display(int *p,int m);
void sumarray(int *p,int n);
int main()
{
int arr[100];
int *ptr,n,i;
ptr=arr;
printf("Enter the number of elments\n");
scanf("%d",&n);
printf("Enter %d elements\n",n);
for(i=0;i<n;i++)
{
        scanf("%d",(ptr+i));
}
printf("The array elements are\n");
display(arr,n);
sumarray(arr,n);
return 0;
}
void sumarray(int *p,int n)
{
        int i,sum=0;
        for(i=0;i<n;i++)
        {
                sum=sum+*(p+i);
        }
        printf("sum of all elements=%d",sum);
}
void display(int *p,int m)
{
        int i;
        for(i=0;i<m;i++)
        {
                printf("%d\n",*(p+i));
        }
}
```

Write a program to pass the elements of an array to a function using pointer and use the function to find the largest element.

```
#include<stdio.h>
void display(int *p,int m);
void largest(int *p,int n);
int main()
{
int arr[100];
int *ptr,n,i;
ptr=arr;
printf("Enter the number of elments\n");
scanf("%d",&n);
printf("Enter %d elements\n",n);
for(i=0;i<n;i++)
{
        scanf("%d",(ptr+i));
}
printf("The array elements are\n");
display(arr,n);
largest(arr,n);
return 0;
}
void largest(int *p,int n)
{
        int large,i;
        large=*p;
        for(i=0;i<n;i++)
        {
                if(*(p+i)>large)
                {
                        large=*(p+i);
                }
        }
        printf("largest element=%d\n",large);
}
void display(int *p,int m)
{
        int i;
        for(i=0;i<m;i++)
        {
                printf("%d\n",*(p+i));
        }
 }
```

Write a program to pass the elements of an array to a function using pointer and use the function to sort them in ascending order.

```
#include<stdio.h>
void display(int *p,int m);
void sort(int *p,int n);
int main()
{
int arr[100];
int *ptr,n,i;
ptr=arr;
printf("Enter the number of elements\n");
scanf("%d",&n);
printf("Enter %d elements\n",n);
for(i=0;i<n;i++)
{
  scanf("%d",(ptr+i));
}
printf("The array elements before sorting are\n");
display(arr,n);
sort(arr,n);
printf("The array elements after sorting are\n");
display(arr,n);
return 0;
}
void sort(int *p,int n)
{
        int i,j,temp;
        for(i=0;i<n-1;i++)
        {
                for(j=0;j<n-i-1;j++)
                {
                        if(*(p+j)>*((p+j)+1))
                        {
                        temp=*(p+j);
                        *(p+j)=*((p+j)+1);
                        *((p+j)+1)=temp;
                        }
                }
        }
}
void display(int *p,int m)
{
        int i;
        for(i=0;i<m;i++)
        {
                printf("%d\n",*(p+i));
        }
}
```

Dynamic memory allocation

What is dynamic memory allocation? Explain the different functions used in dynamic memory allocation?

OR

How dynamic memory allocation can be achieved? Explain with suitable examples.

- The process of allocating and releasing memory at runtime is known as Dynamic memory allocation.
- Using DMA memory space is reserved during program execution and release space when no longer required.

Functions used in Dynamic Memory allocation

There are four library functions: malloc(),calloc(), free() and realloc() are for dynamic memory management. These functions are defined within header file stdlib.h .

1) malloc ()

- ✓ The malloc() function reserves a block of memory of specified size and returns a pointer of type void, which can be casted into pointer of any form.
- ✓ The memory allocation can fail if the space in heap is not sufficient to satisfy the request. If it fails it returns NULL.

Syntax:

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

ptr is pointer of type cast-type.

Example:

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

A memory space equivalent to "100 times the size of integer" is reserved and address of the first memory allocated is assigned to pointer ptr of type int.

2) calloc()

- ✓ calloc() allocates multiple block of storage, each of same size.
- ✓ All bytes are initializes to zero and pointer to the first byte of allocated region is returned.
- ✓ If there is not enough space, a NULL pointer is returned.

Syntax: `ptr=(cast-type*) calloc(n, element-size);`

Example:

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

This statement allocates contiguous space in memory for 25 elements each with size of int.

3) realloc()

This function is used to modify the size of previously allocated space.

Syntax: ptrnew=realloc(ptr,newsize); //Reallocation of space

This function allocates the new memory space of size newsize to the pointer variable ptrnew and returns a pointer to the first byte of new memory.

Sometimes previously allocated space is not sufficient we need to additional space and sometimes allocated memory is much larger than necessary. In both situation, we can change the memory size already allocated with the help of function realloc().

4) free()

It releases the previously allocated space by malloc(), calloc(), and the realloc() function.

Syntax: free(ptr);

Where ptr is a pointer to memory block which has already been created by malloc(),calloc() or realloc() function.

The memory dynamically allocated is not returned to the system until the programmer returns the memory explicitly. This can be done by using free() fuction. This function is used to release the space when it is not required.

Program to dynamically allocate memory for n elements using the calloc() function, then read and display the elements.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int *ptr;
    int i,n;
    printf("Enter  the number of elements\n");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));
    if(ptr == NULL)
    {
      printf("Error! memory not allocated.");
      exit(1);
    }
    printf("Enter %d elements\n",n);
    for(i=0;i<n;i++)
    {
     scanf("%d",(ptr+i));
    }
```

```
        printf("The elements are\n");
        for(i=0;i<n;i++)
        {
          printf("%d\n",*(ptr+i));
        }
        free(ptr);
        return 0;
    }
```

Perform similar operations using malloc()

Hint:

```
ptr=(int*)malloc(n*sizeof(int));
```

Program to illustrate the use of realloc() function

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int *ptr,*ptrnew,i;
ptr= (int*)calloc(2,sizeof(int));
*ptr = 10;
*(ptr+1)=20;
ptrnew =(int *)realloc(ptr, sizeof(int)*3);
*(ptrnew + 2) = 30;
for(i = 0; i < 3; i++)
{
printf("%d ", *(ptrnew+i));
}
return 0;
}
```

Write Program to find sum of n elements in an array using Dynamic memory allocation.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
  int *ptr;
  int i,n,sum=0;
  printf("Enter the number of elements\n");
  scanf("%d",&n);
  ptr=(int*)calloc(n,sizeof(int));
  if(ptr == NULL)
   {
    printf("Error! memory not allocated.");
    exit(1);
   }
```

```
  printf("Enter %d elements\n",n);
  for(i=0;i<n;i++)
  {
          scanf("%d",(ptr+i));
  }
  for(i=0;i<n;i++)
  {
          sum=sum+*(ptr+i);
  }
  printf("The sum of all elements = %d",sum);
  free(ptr);
  return 0;
}
```

Write a program to find the sum of 5 numbers supplied by users using DMA.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
  int *ptr;
  int i,n,sum=0;
  ptr=(int*)calloc(5,sizeof(int));
  if(ptr == NULL)
  {
    printf("Error! memory not allocated");
    exit(1);
  }
  printf("Enter 5 numbers\n",n);
  for(i=0;i<5;i++)
  {
     scanf("%d",(ptr+i));
  }
  for(i=0;i<5;i++)
  {
          sum=sum+*(ptr+i);
  }
  printf("The sum of 5 numbers = %d",sum);
  free(ptr);
  return 0;
}
```

Write a program to sort n elements in an array using dynamic memory allocation.
(Note: Sorting is done in ascending order by default).

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
  int *ptr;
  int i,j,temp,n;
  printf("Enter number of elements\n");
  scanf("%d",&n);
  ptr=(int*)calloc(n,sizeof(int));
  if(ptr == NULL)
  {
    printf("Error! memory not allocated.");
    exit(1);
  }
  printf("Enter %d numbers\n",n);
  for(i=0;i<n;i++)
  {
          scanf("%d",(ptr+i));
  }
  printf("Numbers before sorting are\n");
  for(i=0;i<n;i++)
  {
          printf("%d\n",*(ptr+i));
  }

  for(i=0;i<n-1;i++)
  {
          for(j=0;j<n-1-i;j++)
          {
                  if(*(ptr+j)>*((ptr+j)+1))
                  {
                          temp=*(ptr+j);
                          *(ptr+j)=*((ptr+j)+1);
                          *((ptr+j)+1)=temp;
                  }
          }
  }
printf("Numbers after sorting are\n");
  for(i=0;i<n;i++)
  {
          printf("%d\n",*(ptr+i));
  }
  free(ptr);
  return 0;
}
```

Write a program to read 'n' numbers dynamically and sort them in descending order

Hint:

```
for(i=0;i<n-1;i++)
 {
        for(j=0;j<n-1-i;j++)
        {
                if(*(ptr+j)<*((ptr+j)+1))
                {
                        temp=*(ptr+j);
                        *(ptr+j)=*((ptr+j)+1);
                        *((ptr+j)+1)=temp;
                }
        }
 }
```

Write a program to print reverse element of an array using dynamic memory allocation.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
  int *ptr;
  int i,n;
  printf("Enter number of elements\n");
  scanf("%d",&n);
  ptr=(int*)calloc(n,sizeof(int));
  if(ptr == NULL)
  {
    printf("Error! memory not allocated.");
    exit(1);
  }
  printf("Enter %d elements\n",n);
  for(i=0;i<n;i++)
  {
        scanf("%d",(ptr+i));
  }
  printf("Entered array elements are\n");
  for(i=0;i<n;i++)
  {
        printf("%d\n",*(ptr+i));
  }
  printf("Array elements in reverse order are\n");
  for(i=n-1;i>=0;i--)
  {
        printf("%d\n",*(ptr+i));
  }
  free(ptr);
  return 0;
}
```

Write a program to find the largest and smallest element of an array using DMA.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
  int *ptr;
  int i,n,large,small;
  printf("Enter number of elements\n");
  scanf("%d",&n);
  ptr=(int*)calloc(n,sizeof(int));
  if(ptr == NULL)
  {
    printf("Error! memory not allocated.");
    exit(1);
  }
  printf("Enter %d elements\n",n);
  for(i=0;i<n;i++)
  {
          scanf("%d",(ptr+i));
  }
  printf("Entered array elements are\n");
  for(i=0;i<n;i++)
  {
          printf("%d\n",*(ptr+i));
  }
  large=*ptr;
  small=*ptr;
  for(i=0;i<n;i++)
  {
          if(*(ptr+i)>large)
          {
                  large=*(ptr+i);
          }
           if(*(ptr+i)<small)
           {
                  small=*(ptr+i);
           }
  }
  printf("largest element =%d\n",large);
  printf("smallest element =%d\n",small);
  free(ptr);
  return 0;
}
```

Assignment:

- Write a program to input n numbers in an array and find the sum of all even and odd numbers and count them using DMA.
- Write a program to check if any given number is present in an array or not using DMA.

Write a program to find largest element in an array using dynamic memory allocation and function.

```
#include<stdio.h>
#include<stdlib.h>
void display(int *p,int m);
void largest(int *ptr,int y);
int main()
{
    int *ptr;
    int i,n;
    printf("Enter the number of elements\n");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));
     if(ptr == NULL)
     {
         printf("Error! memory not allocated.");
          exit(1);
      }
      printf("Enter %d elements\n",n);
      for(i=0;i<n;i++)
      {
      scanf("%d",(ptr+i));
      }
      printf("The elements are\n");
      display(ptr,n);
      largest(ptr,n);
      free(ptr);
}
void display(int *p,int m)
{
         int i;
         for(i=0;i<m;i++)
         {
                  printf("%d\n",*(p+i));
         }
}
void largest(int *ptr,int y)
{
         int i, large;
         large=*ptr;
         for(i=0;i<y;i++)
         {
                  if (large<*(ptr+i))
                  {
                           large=*(ptr+i);
                  }
         }
         printf("largest element=%d\n",large); }
```

Memory leak

A memory leak occurs when a block of memory that was previously allocated by a programmer is not properly de-allocated by the programmer. Even though that memory is no longer in use by the program, it is still "reserved", and that block of memory cannot be used by the program until it is properly de-allocated by the programmer. That's why it's called a memory leak. To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

```
/* Function without memory leak */
#include <stdlib.h>
 int main()
{
int *ptr = (int *) malloc(sizeof(int));
/* Do some work */
free(ptr);
return 0;
}
```

How dynamic memory allocation is better than static memory allocation?

In c programming, Memory can be allocated in two ways.

They are:

1) **Static Memory allocation**

   In static memory allocation the memory is allocated before the execution of program begins (During compilation).In this type of allocation the memory cannot be resized after initial allocation. So it has some limitations. Like

   - Wastage of memory
   - Overflow of memory

   eg. int arr[100];

   Here, the size of an array has been fixed to 100.If we just enter to 10 elements only, then there will be wastage of 90 memory location and if we need to store more than 100 elements there will be memory overflow.

2) **Dynamic memory allocation**

   To overcome the limitation of static memory allocation, dynamic memory allocation technique is used. In this type of memory allocation the memory is allocated during execution of program, so that we can allocate and release memory as per requirement during runtime. Functions calloc(),malloc(),realloc() and free() are used in DMA which are defined in stdlib.h header file .

   Let's use the calloc() function to allocate memory for n integer values:

```
int *ptr;
ptr = (int *)calloc(n, sizeof(int));
```

   Here, we allocate memory for n integer values at runtime as needed, preventing underflow and overflow of memory.

Differentiate between static memory allocation and dynamic memory allocation.

Static memory allocation	Dynamic memory allocation
1. Memory is allocated before the execution of program begins.(During Compilation)	1. Memory is allocated during the execution of program.
2. Variable remain permanently allocated.	2. Allocated only when program unit is active.
3. In this type of allocation memory cannot be resized after initial allocation.	3. In this type of allocation memory can be dynamically expanded and shrunk as necessary.
4. Implemented using stacks.	4. Implemented using heap.
5. Faster than dynamic memory allocation.	5. Slower than static memory allocation.
6. It is less efficient than dynamic memory allocation strategy.	6. It is more efficient than static memory allocation strategy.
7. Memory cannot be reuse when it is no longer needed.	7. Memory can be freed when it is no longer needed and reuse and reallocate during execution.

Advantages of dynamic memory allocation

- Dynamic Memory allocation is done at runtime.
- No need-to-know amount of memory prior to allocation
- We can create additional storage whenever we need them.
- We can de-allocate (free/delete) dynamic space whenever we are done with them.
- No wastage of memory
- No shortage of memory.

Disadvantages of dynamic memory allocation

- Slower execution.
- Memory needs to be freed.

Write short notes on:

Dynamic memory management /Dynamic memory allocation

Dynamic memory management refers to managing system memory at runtime. This memory management technique allows us to allocating and releasing memory during runtime. There are four library functions that are defined in header file <stdio.h> for dynamic memory allocation.

Function	Syntax	Use of function
malloc()	ptr=(cast-type*)malloc(byte-size)	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	ptr=(cast-type*)calloc(n,element-size);	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
realloc()	ptr=realloc(ptr,newsize);	Change the size of previously allocated space
free()	free(ptr);	De-allocate the previously allocated space

Let's use of calloc() function to allocate memory dynamically

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

This statement allocates contiguous space in memory for 25 elements each with size of int.

malloc() vs calloc()

Malloc	Calloc
1. malloc() allocates memory block of given size (in bytes)	1. calloc() allocates a region of memory large enough to hold "n elements" of "size" bytes each.
2. malloc() doesn't initialize the allocated memory.	2. The allocated region is initialized to zero.
3. It takes one argument and allocates the memory in bytes given in argument.	3. The calloc() function takes two arguments number of variables to be allocated and size of each variable.
4. Syntax: ptr=(cast-type*)malloc(byte-size);	4. Syntax: ptr=(cast-type*)calloc(n,element-size);
5. eg. int *ptr; ptr=(int*)malloc(100*sizeof(int)); A memory space equivalent to "100 times the size of integer" is reserved	5. eg. int *ptr; ptr=(int*) calloc(25,sizeof(int)); This statement allocates contiguous space in memory for 25 elements each with size of int.

Below are the different definitions of the function **search()**

i)
```
void search(int *m[],int x)
{
}
```

ii)
```
void search(int *m,int x)
{
}
```

Are they equivalent? Explain.

Solution:

In function definition,

```
void search(int *m ,int x)
{
}
```

Here, pointer is used as an function argument, which has been used to accept the address of integer type element. Here, we can pass address of one element at a time. In case of array beginning address of array is passed.

But, in function definition

```
void search(int *m[],int x)
{
}
```

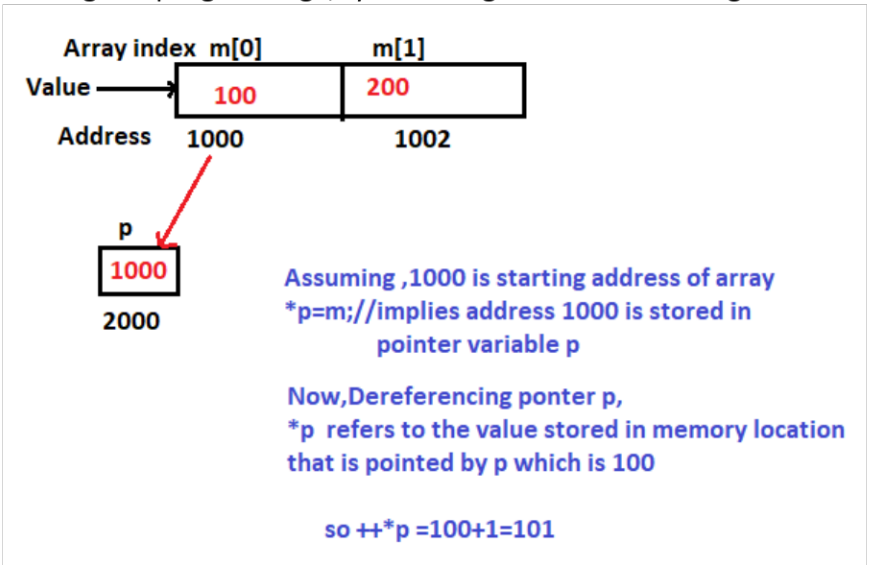
When array is passed as an argument to function, the address of entire array can be passed. Which means array of pointers to an int.

Write the output:

```
void main()
{
int m[2];
int *p=m;
m[0]=100;
m[1]=200;
printf("%d%d",++*p,*p);
}
```

Solution:

Tracing the program logi ,by assuming 1000 is an starting address of an array



Output :

```
101
100
```

3) Find the output:

```
void fun(int *p);
void main()
{
        int x=4;
        printf("%d\n",x);
        fun(&x);
        printf("%d\n",x);
}
void fun(int *p)
{

*p=*p/2+13;
}
```

(Trace the program yourself)

Output:

```
4
15
```

4) **Trace the output of the following program.**

```
#include<stdio.h>
int main()
{
int x=25;
int *y;
int **z;
y=&x;
z=&y;
printf("x=%d\n",++x);
printf("y=%d\n",*y);
printf("z=%d\n",**z++);
return 0;
}
```

(Trace the program yourself)

Output:

```
x=26
y=26
z=26
```

5) Trace the output of the following program.

```
void main()
{
int arr[]={10,20,30,45,67,58,74}
int *i,*j;
i=&arr[1];
j=&arr[5];
printf("%d\n%d",j-i,*j-*i);
}
```

(Trace the program yourself)

Output

```
4

38
```

6) Trace the output of the following program.

```
#include<stdio.h>
int main()
{
float a[5]={13.24,1.5,1.5,5.4,3.5};
float *j,*k;
j=&a[0];
j=j+4;
k=&a[2];
printf("\n%.2f\n%.2f\n%.2f\n%d",*j,*k,*j-*k,j-k);
}
```

(Trace the program yourself)

Output:

```
3.50
1.50
2.00
2
```

7) Trace the output of the following program.

```
#include<stdio.h>
void main()
{
int x,y,z,k;
int *ptr;
x=10;
ptr=&x;
printf("%d\n",x);
printf("%d\n",*ptr);
z=*ptr+1;
k=++(*ptr);
printf("%d\n",k);
printf("%d\n",*ptr);
printf("%d",z);
}
```

(Trace the program yourself)

Output

```
10
10
11
11
11
```

8) **Trace the output of the following program**.

```
void main()
{
        float f[]={10.5,1.5,2.5,3.5,4.5};
        float *p,*q;
        p=f;
        p=p+4;
        q=&f[2];
        printf("%f\t%f\t%f",*p,*q,*p-*q);
}
```

(Trace the program yourself)

Output:

```
4.50000         2.500000        2.000000
```