Pointer

Every variable is stored in some memory location. This location is called address of the variable. To store such address, we need a pointer. So a pointer is a variable or constant that is used to store the address.

A variable may contain any type of data; its address is always an integer.

Declaration of a pointer variable:

<datatype> * pointer;

Ex: int * p;

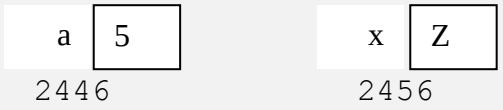
char * q;

Here p is called a 'pointer to integer' or integer pointer, which can hold the address of an int type variable. Similarly q is a pointer to char, that can store the address of a character variable.

Operators

1. Reference operator (&) : It returns the address of a variable.
2. De-reference operator/indirection operator(*): It returns the value stored at an address.

```
int a = 5;
char x='Z';
int *p;
char *q;
p = &a;
q = &x;
printf ("\n%d \t %c",*p,*q);
printf ("\n%p \t %p",p,q);
```

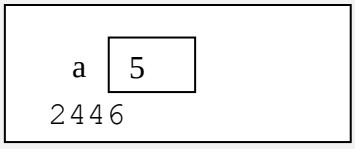



```
Output
5     Z
2446 2456
```

Pointer arithmetic

1. **Pointer + int:** Here operator + is a move forward operator. This expression P + x returns x location ahead of the location pointed by p.

```
int a=5;
int  *p=&a[0];
int *q;
q=a+6;
printf("%p",q);
```



2. **Pointer - int:** Here operator - is a move backward operator. The expression P - x returns x location before of the location pointed by p.

```
printf("%p",q-2);
```

3. **Pointer – pointer:** This operation returns the number of location between two pointers.

```
printf("%d",q-p);
```

4. **Pointer ++ :** This operation makes the pointer to point the next location.

```
p++;
printf("%p",p);
```

5. **Pointer --** : This operation makes the pointer to point the previous location
6. **The relational operators (<,>,<=,>=,!=,==):**

   Relational operators can be performed to compare two pointer of same type.

```
printf("%d",p<q);
```

7. **Pointer + pointer:** This operation is not allowed.

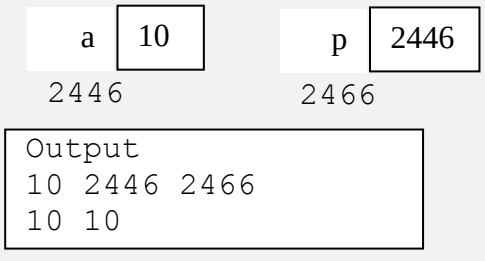
Pointer to pointer:

When a pointer holds address of another pointer variable, it is called as pointer to pointer.

Syntax: <DataType> **Pointername;

Example:

```
int a=10;
int *p;  //p is declared as pointer to int
int **q //q is declared as pointer to pointer to int
p = &a;
q = &p;
printf("\n%d %p %p", a,p,q);
printf("\n%d %d", *p,**q);
```



Void pointer :

Void pointer is a generic pointer. It has no specific type. It can hold the address of any type of variable. However, it needs to be type casted to a specific type pointer to get the value from it.

```
void main()
      {
      void *ptr;
```

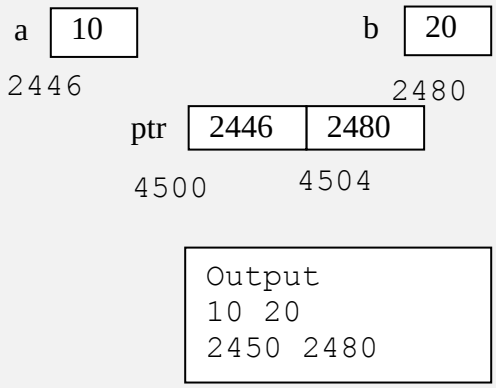
```
int a=6;
ptr=&a;
printf("%d" , *ptr) ;              //!! Error
printf("%d",  * (int *)ptr);       //  Now the output is 6
}
```

Array of Pointer:

If an array contains multiple numbers of addresses in it, it is a called array of pointers.

Syntax: <DataType> *ArrayName[Size];

```
int main()
    {
    int *ptr[2];
    int a=10,b=20,i;
    ptr[0]=&a;
    ptr[1]=&b;
    printf("\n%d %d ", **(ptr+1),**ptr);
    printf("\n%d %d", *ptr+1,*(ptr+1));
    }
```



Pointer to Function:

Like a variable, a function also has a memory address. The pointer, that points to the address of function is called pointer to function. Unlike a general pointer, a pointer to function does not point to the data. It points to the first line code of a function.

Declaration of pointer to function

<returnType> (*pointerToFunctionName) (ArgumentType);

The declaration of pointer to function must matches to the function it will point to.

Function	Pointer to function
int fun(int a , int b);	int (*ptr)(int,int);
void fun(int *p);	void (*ptr)(int *);
double *fun(double x, double y);	double * (*ptr)(double, double);
int fun(char s[],int n)	int (*ptr)(char [],int)

Assigning address of the function to its pointer.

We can get the address of a function by using only its name. So function name without any brace is used to assign its address to its pointer. See the format below.

Syntax: pointerToFunction = FunctionName ;

```
ptr = fun;
```

Invoking the function through pointer to function

A function can also be invoked by using its pointer.

variable = pointerToFunction(argument1,argument2,…..);

Consider following function.

```
int fun(int a,int b)
      {
      return a + b;
      }
```

Following are the steps to call a function through its pointer

- Declare the pointer to function: int (*ptr)(int,int);
- Assign the address of function : ptr = fun ;
- Call the function through the pointer: k = ptr (10,20);

Program: Write a program to input two numbers and find the sum. Use pointer to function to call the addition function

```
#include<stdio.h>
int addition(int,int);
void main()
      {
      int a,b,sum;
      int (*ptr)(int, int);          // declaration  of Pointer to Function
      ptr = addition;                //Address of function is assigned to pointer
      printf("Enter two numbers:");
      scanf("%d%d",&a,&b);
      sum = ptr(a,b);                //call to the addition() through pointer
      printf("\nSum of two numbers = %d",sum);
```

```
    }
//Function definition
int addition(int a, int b)
    {
    return a + b;
    }
```

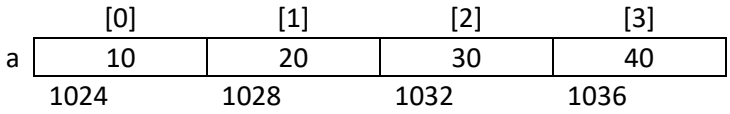
Pointer and Array

Pointer and array are closely related to each other. All operations on array can be performed using pointer and also a pointer can also be used to store and manipulate multiple number data as we did in array.

Array name VS pointer to first element of array:

Base address of an array is the address of first element of the array. The base address is referred by the name of the array. Array name is a pointer constant. So array name and the pointer that points to address of first element of the array are same; they represent the same memory location.

Only difference is that array name is a **pointer constant.** But a pointer that points to the address of 1st element of array is not a constant. We can alter the address it points to.




```
int a[ ] = {10,20,30,40};   // An array 'a' is declared.
int * p;
p=&a[0];          //Pointer 'p' points to address of a[0]
printf("%p  %p",a,p);
```

Here, printf () will print the same address. So the array name 'a' can be used in place of a pointer.

Name of the array can also be de-referenced, as a pointer is de-referenced.

```
printf ("%d ", *a);
```

Program: Following program shows how an array name is used as pointer. It prints all elements of an array by de-referencing the array name.

```
#include<stdio.h>
int main()
   {
   int a[]={10,20,30,40},i;
   for(i=0;i<4;i++)
        printf(" %d",*(a+i));
   return 0;
   }
```

A pointer can also be used to access elements of the array. There are two ways to print elements of an array through pointer. For example let p is a pointer to the base address of an array, then *(p+1) and p [1], both refers to the element present at index 1 in the array.

Program: Following program show how the data are inserted into an array of n elements and the printed using pointer.

```
#include<stdio.h>
int main()
   {
   int a[40],n,i;
   int *p;
   p=a;   //Pointer points to the base address
   printf("\nEnter number of data:");
   scanf("%d",&n);
   printf("\nEnter data in the array:");
   for(i=0;i<n;i++)
       {
       scanf(" %d",(p+i));
       }
   /*print the data */
   for(;p<a+n;p++)
       printf(" %d",*p);
return 0;
}
```

Exercise: Write a program to sort an array of n elements using pointer.

Pointer to Array

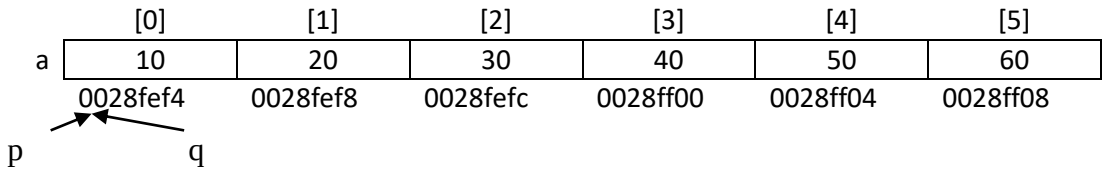
Just we see how a pointer points to the address of an array element. A pointer can also point to entire array. This is called pointer to array. The declaration of pointer to array is as follow

Syntax: datatype (*pointername)[size]

Example: `int (*p)[5];`

Here the pointer to array 'p' can point to the an array of 5 integer. See following code to have clear understanding of the difference between pointer, that points to base address and pointer to entire array, which points to entire array.

```
#include<stdio.h>
int main()
        {
        int a[]={10,20,30,40,50,60};
        int *p;
        int (*q)[6];
        p=a;
        q=&a;
        :
```



In above program, 'p' is a pointer that points to the address of 0th element of array.'q' is a pointer to entire array. We can see both points to same address. But they are different by their size and content.

```
printf("%p %p",p,q);
```

```
Output
0028fef4 0028fef4
```

This will surely print the same address, ie, 0028fef4 .output of this line is:

```
printf("\n%d %p",*p,*q);
```

```
Output
10 0028fef4
```

When we de-reference these pointers, we get the value stored at the address pointer by these pointers. So dereferencing 'p', we get an int value, same as *a, and also a[0]. De-referencing 'q', we get an array, means the base address of array, which is itself a pointer.

```
printf("\n%d %d",sizeof(*p),sizeof(*q));
```

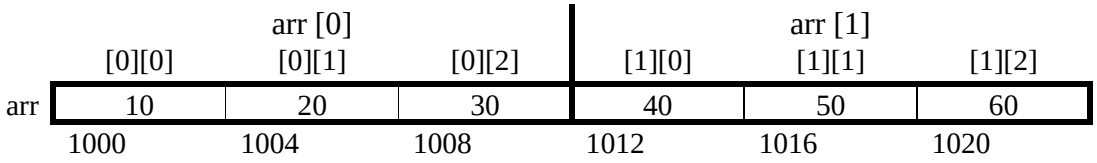
```
Output
4 24
```

Pointer and Two-Dimensional Array

Like an array, the base address of a two-dimensional array is also a pointer constant. But here, when we de-reference it, we do not get integer. We get an array of integers. Because the 0th element in two-dimensional is another array. Consider following code and its memory allocation.

```
int arr[2][3] = {{10,20,30},{40,50,60}};
```

Here 'ptr' is a two–dimensional array, which consists of two one-dimensional arrays of three integers.



Here arr[0] points to the 0th element, which is itself an array. So arr[0] is same as *(arr+0).

To print any element in normal way, we use double subscript notation. For example arr[i][j] represent the jth element in ith array in 2D array 'arr'.

```
printf("%d", *arr[0]);
printf("%d", *(*(arr+0)+0));
```

Above two statement represent the 0th element in 0th array i,e arr[0][0] which is 10. So *(*(arr + i) + j) is used to represent an element in ith row and jth column.

Program: Program to print elements of a two-dimensional array using pointer.

```
#include<stdio.h>
int main()      {
        int arr[2][3]={{10,20,30},{40,50,60}};
        int i,j;
        for(i=0;i<2;i++)
                {
                for(j=0;j<3;j++)
                        printf("\t%d",*(*(arr+i)+j));
                printf("\n");
                }
        return 0;
        }
```

```
Output
10    20    30
40    50    60
```

Program: Input a matrix pXq matrix A and another matrix B of size qXr. Multiply two matrixes and show the resulting matrix.

```
#include<stdio.h>
int main()	{
	int A[20][20],B[20][20],C[20][20];
	int i,j,k,p,q,r;
	printf("Enter row and column size of first matrix:");
	scanf("%d%d",&p,&q);
	printf("Enter data into first matrix of size %d x %d:",p,q);
	for(i=0;i<p;i++)		//Input First matrix
		{
		printf("\nIn row %d:",i);
		for(j=0;j<q;j++)
			scanf("\t%d",(*(A+i)+j));
		}
	printf("Enter column size of 2nd matrix:");
	scanf("%d",&r);
	printf("Enter data into second matrix of size %d x %d:",q,r);
	for(i=0;i<q;i++)		//Input Second matrix
		{
		printf("\nIn row %d:",i);
		for(j=0;j<r;j++)
			scanf("\t%d",(*(B+i)+j));
		}
	//Multiplying two matrices
	for(i=0;i<p;i++)
		{
		for(j=0;j<r;j++)
			{
			*(*(C+i)+j)=0;
			for(k=0;k<q;k++)
				*(*(C+i)+j)=*(*(C+i)+j)+*(*(A+i)+k)* *(*(B+k)+j);
			}
		}
	//print the resulting matrix
	printf("\nFirst Resulting matrix:\n");
	for(i=0;i<p;i++)
		{
		for(j=0;j<r;j++)
			printf("\t%5d",*(*(C+i)+j));
		printf("\n");
		}
	return 0;
	}
```

Pointer and String

Like array, we can also manipulate strings using pointer. A string name is a base address of the string that points to the address of first character. See the code below.

```
char str[ ]=”ABCDE”;
printf(“%s”,str);
```

Here the string name ‘str’ is a constant pointer that points to the address of first character of string. Format string %s in printf () function prints all characters stored in the address starting from address pointed by ‘str’ to the end of string. So it is important to understand following code.

```
printf(“%s”,str+0);
printf(“%s”,str+1);
```

As (str+0) points to same address as str points, so above code also prints same characters as printf (“%s”,str); prints. Second printf () prints all characters, starting from the address (str+1) to end of the string. See the following code segments for more details understanding.

Program: Initialize a string and print the pattern like following specification

```
Input:   ABCDE
Output:  ABCDE
         BCDE
         CDE
         DE
         E
```

```
#include<stdio.h>
int main(){
        char str[20];
        int i;
        printf("Input a String:");
        scanf("%s",str);
        for(i=0;str[i]!='\0';i++)
                printf("\n%s",str+i);
        return 0;
        }
```

Program: Initialize a string. Then print all the characters in different way.

```
#include<stdio.h>
int main()
	{
	char str[]="Save World";
	char * p;
	int i;
	// print the string using string name as pointer
	for(i=0;*(str+i)!='\0';i++)
		{
		printf(" %c",*(str+i));
		}
	// print the string using pointer to string
	for(p=str;(*p)!='\0';p++)
		{
		printf(" %c",*p);
		}
	return 0;
	}
```

Exercise: Input a text and a pattern. Search the pattern in text and print the location if found.

Dynamic Memory allocation

Allocating the memory spaces for program is called memory allocation. Memory allocation is performed by operating system either statically of dynamically.

In **static memory allocation**, the size of memory space required by the program is specified, when the program is written. The programmer is well aware of the size of memory space.

```
int arr[60];                    // Static memory allocation
```

In above code, operating system is requested to reserve the memory space for 60 integer type of data.

The disadvantage of this scheme is:

1. If we need to store more than 60 numbers in the array, we will not able to store data more than 60.
2. If we need to store only 5 numbers, then a lot of memory space is unnecessary blocked and left unused.

To overcome this situation, dynamic memory allocation is used.

In **dynamic memory allocation**, size of memory spaces to be allocated is specified at the time of execution of the program. Here, the user of the program specifies the size of memory space as per the requirement.

Following functions defined in <stdlib.h>, are used to allocate and de-allocate the memory spaces at run time.

1. malloc() : This function reserves the memory spaces at the time execution of a program. It allocates a single block of memory location of required size and returns the base address of allocated memory. The return type of this function is void *

Syntax: `PointerVariable =(DataType *) malloc(SizeOfMemoryinBytes);`

2. calloc() : This function reserves the memory spaces at the time execution of a program. It allocates multiple blocks of adjacent memory and returns the base address of allocated memory. The return type of this function is also void *

Synatx: `PointerVar =(DataType *)calloc(NumberOfBlocks,SizeOfBlock);`

3. realloc(): This function changes the size of an already allocated memory block.This can extend and reduce the allocated memory size.

```
PointerVariablere = realloc(pointervariable,newSizeinbytes);
```

4. free () : This function releases the memory location that no longer needed.

```
free(pointervariable);
```

Program: This program shows use of malloc() and free().The program dynamically allocates accepts an array to accepts a set of data from user. Then calculate and print the sum of all the data.

```
#include<stdio.h>
#include<stdlib.h>
void main()    {
     int *p, i, n, sum=0;
     printf("Enter  number of data you want:");
     scanf("%d",&n);
     p=n(int *) malloc(n * sizeof(int));
     for(i=0;i<n;i++)
          {
          printf("Enter the data:");
          scanf("%d",&p[i]);
          }
     for(i=0;i<n;i++)
          Sum=sum + p[i] ;
     printf("\nSum=%d",sum);
     free(p);
     }
```

Output:

```
Enter number of data you want:3
Enter the data:66
Enter the data:40
Enter the data:34
Sum=140
```

Program: This program shows use of calloc() and free().The input the data in an array and in a sub-array and a position to insert the sub-array into the array. Two arrays are dynamically allocated using calloc().The program then use realloc() to extend the memory space of the array to accommodate the sub-array. Then print the extended array.

```
#include<stdio.h>
#include<stdlib.h>
int main()
     {
     int *arr,*subarr;
     int n1,n2,i,j,pos;
     //input size of array
     printf("Enter size of the array:");
     scanf("%d",&n1);
     //allocating memory to the array
     arr=(int *) calloc(n1,sizeof(int));
     //input data in array
     for(i=0;i<n1;i++)
          scanf("%d",(arr+i));
     //input size of sub-array
     printf("Enter size of sub-array:");
     scanf("%d",&n2);
     //allocating memory to the sub-array
     subarr=(int *) calloc(n2,sizeof(int));
     //input data in sub-array
     printf("Enter the data in sub-array:");
     for(i=0;i<n2;i++)
          {
          scanf("%d",(subarr+i));
          }
     //Asking the position
     printf ("Enter position to insert sub-array in array:");
     scanf("%d",&pos);
     //Reallocate the array with extended memory
     arr = realloc (arr,(n1+n2)*sizeof(int));
     //code to insert the subarray in array
     for(i=n1-1;i>=pos;i--)
          {
          *(arr+i+n2) = *(arr+i);
          }
     n1=n1+=n2;
     j=pos;
     for(i=0;i<n2;i++)
          {
```

```
        *(arr+j) = *(subarr+i);
        j++;
        }
        //print the sub-array
    for(i=0;i<n1;i++)
        printf(" %d",*(arr+i));
    free(arr);
    free(subarr);
    return 0;
    }
```

Output:

```
Enter size of the array:4
Enter the data in array:1 2 3 4
Enter size of the sub-array:6
Enter the data in sub-array:80 70 50 40 30 20
Enter the position (index) to insert the sub-array in the array:2
1 2 80 70 50 40 30 20 3 4
```