Pointers in C

Pointer:

- Pointer is derived data type.
- Primitive variable stores value (data).
- Pointer variable stores address.

Syntax:

```
datatype*  name;
```

Example:

```
int*  p;
   or
int  *p;
```

Declarations:

```
int* p, q ;   --> only 'p' is pointer and 'q' is integer type
int  *p, *q ;  -->  both p and q are pointers.
```

Classification: Pointers is of 2 types

Typed Pointers:

- Typed pointer always points to specific type of data.
    - int* -> points to int data
    - float* -> points to float data
    - struct Emp* -> points to Emp data

Untyped:

- Untyped pointer can points to any type of data. It is also called Generic pointer
    - void* -> any data

Operators:

- The 2 operators are used to process the information using pointers in C.
- "&": Address operator, returns the memory address of specified location.
- "*": Pointer operator, returns the data inside specific address.

```
#include<stdio.h>
int main()
{
        int i=10;
        int* p=&i;
        printf("%d \n", i);
        printf("%d \n", &i);
        printf("%d \n", p);
        printf("%d \n", &p);
        printf("%d \n", *p);
        printf("%d \n", *(&i));
        printf("%d \n", *(&p));
        printf("%d \n", **(&p));
```

```
	return 0;
}
```

Call by value:

- Calling the function by passing values as parameters.
- We collect these parameters into arguments of function.
- Arguments - local variables.
- Arguments allowed to access only from the same function.
- Process data using arguments cannot be accessed from calling function.

```
#include<stdio.h>
void swap(int,int);
int main()
{
	int a=10, b=20;
	swap(a,b);
	return 0;
}
void swap(int x, int y)
{
	int temp;
	temp = x;
	x = y;
	y = temp;
	printf("After swap : %d, %d\n", x, y);
}
```

Call by reference:

- "Reference" is nothing but "address".
- Calling the function by passing address as input.
- Only a pointer can store address.
- We need to collect these addresses into pointer type arguments in function.
- Processed information must be accessed using pointers from calling function.

```
#include<stdio.h>
void swap(int*, int*);
int main()
{
	int a=10, b=20;
	swap(&a, &b);
	printf("After swap : %d, %d\n", a, b);
	return 0;
}
void swap(int* x, int* y)
{
	int temp;
	temp = *x;
	*x = *y;
	*y = temp;
```

```
        printf("After swap : %d, %d\n", *x, *y);
}
```

- Passing Array or String are also the examples of Call by reference.
- Array variable stores base address of memory block.
- Passing array is nothing but passing base address.
- We collect the address into same type of array variable as an argument.

```
#include<stdio.h>
void modify(int[]);
int main()
{
        int a[5]={10,20,30,40,50}, i;
        modify(a);
        for(i=0 ; i<5 ; i++)
        {
                printf("%d \n", a[i]);
        }
        return 0;
}
void modify(int x[])
{
        int i;
        for(i=0 ; i<5 ; i++)
        {
                x[i] = x[i]+5;
        }
}
```

Size of variable:

- Variable stores data.
- The size of variable depends on the data we store.

```
#include<stdio.h>
int main()
{
        char a;
        short b;
        float c;
        int d[5];

        printf("char size : %d \n", sizeof(a));
        printf("short size : %d \n", sizeof(b));
        printf("float size : %d \n", sizeof(c));
        printf("array size : %d \n", sizeof(d));
}
```

Size of Pointer:

- Pointer variable stores address.

- Address(memory location) is an integer.
- Integer size varies from compiler to compiler
    - 16 bit --> 2 bytes
    - 32 bit --> 4 bytes

```
#include<stdio.h>
struct Emp
{
	int no;
	char name[20];
	float salary;
};
int main()
{
	char* a;
	short* b;
	float* c;
	struct Emp* d;

	printf("char* size : %d \n", sizeof(a));
	printf("short* size : %d \n", sizeof(b));
	printf("float* size : %d \n", sizeof(c));
	printf("Emp* size : %d \n", sizeof(d));
}
```

Pointers modify:

- Modify operators are ++ , --
- Modify operators increase or decrease the value of variable by 1.

```
#include<stdio.h>
int main()
{
	int x=10, y=20;
	printf("++x value : %d \n", ++x);
	printf("--y value : %d \n", --y);
}
```

- When we modify pointer, the value either increase or decrease by "size" bytes.
- char* value increase by 1
- float* value increase by 4

```
#include<stdio.h>
int main()
{
	char c = 'g';
	short s = 10;
	double d = 23.45;

	char* p1 = &c;
	short* p2 = &s;
```

```
        double* p3 = &d;

        printf("p1 val : %d \n", p1);
        printf("++p1 val : %d \n", ++p1);

        printf("p2 val : %d \n", p2);
        printf("++p2 val : %d \n", ++p2);

        printf("p3 val : %d \n", p3);
        printf("++p3 val : %d \n", ++p3);
}
```

- Pointer modify is useful when we process elements of array using pointers.
- When we modify the address of array, it is pointing to next location of array to access that element.
- Generally we access array elements using index starts with 0 to size-1.

```
#include<stdio.h>
int main()
{
        int arr[5] = {10,20,30,40,50}, i;
        printf("Array elements are :\n");
        for(i=0 ; i<5 ; i++)
        {
                printf("%d \n", arr[i]);
        }
        return 0;
}
```

- Array variable is implicit pointer variable.
- Array variable holds address.
- When we specify index, implicitly it uses pointers to access the elements.
- arr[i] implicitly converts into *(arr+i)

```
#include<stdio.h>
int main()
{
        int arr[5] = {10,20,30,40,50}, i;
        printf("Array elements are :\n");
        for(i=0 ; i<5 ; i++)
        {
                printf("%d \n", *(arr+i));
        }
        return 0;
}
```

Pointer to array: When a pointer pointing to array, we can process elements in many ways using expressions:

```
#include<stdio.h>
int main()
{
	int arr[5] = {10,20,30,40,50}, i;
	int* ptr = arr;
	printf("Array elements are :\n");
	for(i=0 ; i<5 ; i++)
	{
		printf("%d,%d,%d,%d\n", *(ptr+i), *(i+ptr), ptr[i], i[ptr]);
	}
	return 0;
}
```

String processing:

```
#include<stdio.h>
int main()
{
	char s[5] = "amar";
	int i;
	for(i=0 ; i<4 ; i++)
	{
		printf("%c,%c,%c,%c \n", *(s+i), *(i+s), s[i], i[s]);
	}
	return 0;
}
```

- We need to consider the priority of operators while accessing array elements using pointers.
- First priority : ++, --
- Second priority : pointer(*)
- Third priority : arithmetic operators
- Note : ( ) having higher priority.

```
#include<stdio.h>
int main()
{
	int arr[5] = {10,20,30,40,50},i;
	int* ptr;
	ptr = arr;
	printf("%u\n", *++ptr + 3);
	printf("%u\n", *(ptr-- + 2) + 5);
	printf("%u\n", *(ptr+3)-10);
	return 0;
}
```

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

```
	int arr[5] = {8, 3, 4, 9, 2},i;
	int* ptr;
	ptr = arr;
	printf("%u\n", *(--ptr+2) + 3);
	printf("%u\n", *(++ptr + 2) - 4);
	printf("%u\n", *(ptr-- +1 ) + 2);
	return 0;
}
```

Pointer to String: We represent strings using character arrays.

```
#include<stdio.h>
int main()
{
	char s[5] = "five";
	printf("%s \n", s);
	return 0;
}
```

- char* can points to single character as well as string.
- it will consider the size based on assigned string value.

```
#include<stdio.h>
int main()
{
	char* s = "online-c";
	printf("%s \n", s);
	return 0;
}
```

```
#include<stdio.h>
int main()
{
	char* s = "ONLINE";
	printf("%s \n", s);
	printf("%c \n", s);
	printf("%c \n", *s);
	printf("%c \n", *s+3);
	printf("%c \n", *(s+3));
	return 0;
}
```

```
#include<stdio.h>
void main()
{
	char* str = "learnown";
	printf("%c\n", *str++ + 3);
	printf("%s\n", ++str+2);
}
```

```
#include<stdio.h>
void main()
{
	char* str = "learnown";
	printf("%c\n" , *(str++ + 2)+3);
	printf("%c\n" , *++str+2);
	printf("%s\n" , --str-1);
}
```

```
#include<stdio.h>
void main()
{
	char* str = "learnown";
	printf("%c\n",*((str-- +2)+1)-3);
	printf("%c\n", *(--str + 3)-32);
	printf("%c\n",*(++str+2)+4);
}
```

```
#include<stdio.h>
void main()
{
	char sport[ ]= "cricket";
	int x=1 , y;
	y=x++ + ++x;
	printf(“%c”,sport[++y]);
}
```

Pointer to Structure:

- We can create pointers to user defined data types also.
- When we access the elements of structure, we use dot(.) operator.

```
#include<stdio.h>
struct Emp
{
	int no;
	char name[20];
	float salary;
};
int main()
{
	struct Emp e = {101, "amar", 30000};
	printf("No : %d \n", e.no);
	printf("Name : %s \n", e.name);
	printf("Salary : %f \n", e.salary);
	return 0;
}
```

When a pointer is pointing to structure, we use arrow(->) operator.

```
#include<stdio.h>
struct Emp
{
```

```
	int no;
	char name[20];
	float salary;
};
int main()
{
	struct Emp e = {101, "amar" , 30000};
	struct Emp* p = &e;
	printf("No : %d \n", p->no);
	printf("Name : %s \n", p->name);
	printf("Salary : %f \n", p->salary);
	return 0;
}
```

Array of pointers:

- Pointer stores address.
- Array of pointers means, "An array that stores addresses".
- Assigning references and accessing elements using index :

```
#include<stdio.h>
int main()
{
	int a[5]={10,20,30,40,50};
	int* p[5];
	for(i=0 ; i<5 ; i++)
	{
		p[i] = &a[i];
	}
	printf("Elements : \n");
	for(i=0 ; i<5 ; i++)
	{
		printf("%d\n", *p[i]);
	}
	return 0;
}
```

Assigning references and accessing elements using pointers:

arr[i] ---> *(arr+i)

```
#include<stdio.h>
int main()
{
	int a[5]={10,20,30,40,50};
	int* p[5];
	for(i=0 ; i<5 ; i++)
	{
		*(p+i) = a+i;
	}
	printf("Elements : \n");
	for(i=0 ; i<5 ; i++)
```

```
	{
		printf("%d\n", **(p+i));
	}
	return 0;
}
```

Array of Strings:

- char* can holds a string.
- Array of char* variable can hold more than one string.

```
#include<stdio.h>
void main()
{
	char* s[] = {"abc", "xyz", "lmn", "pqr"};
	int i;
	printf("Strings are : \n");
	for(i=0 ; i<4 ; i++)
	{
		printf("%s \n", s[i]);
	}
}
```

```
#include<stdio.h>
void main()
{
	char* s[] = {"abc", "xyz", "lmn", "pqr"};
	int i;
	printf("Strings are : \n");
	for(i=0 ; i<4 ; i++)
	{
		printf("%s \n", *(s+i));
	}
}
```

```
#include<stdio.h>
void main()
{
	char* s[] = {"abc", "xyz", "lmn", "pqr"};
	int i;
	printf("Strings are : \n");
	for(i=0 ; i<4 ; i++)
	{
		/* printf("%c \n", s[i]); */
		printf("%c \n", *s[i]);
	}
}
```

```
#include<stdio.h>
void main()
{
	char* s[] = {"abc", "xyz", "lmn", "pqr"};
	int i;
	printf("Strings are : \n");
	for(i=0 ; i<4 ; i++)
	{
		printf("%c \n", *(*(s+i)+1));
	}
}
```

Pointer to pointer:

- It is also called double pointer.
- Pointer always holds address of variable.
- Pointer to Pointer holds address of another pointer variable.

```
#include<stdio.h>
void main()
{
	int x = 10;
	int* ip = &x;
	int** ipp = &ip;

	printf("%d \n", x);
	printf("%d \n", *ip);
	printf("%d \n", **ip);
}
```

```
#include<stdio.h>
void main()
{
	char* s[] = {"abc","cde","efg"} ;
	int i ;
	for(i=0 ; i<3 ; i++)
	{
		printf("%c \n", *(*(s+i)+i));
	}
}
```

```
#include<stdio.h>
void main()
{
   char *s[ ] = {"black", "white", "pink", "violet"};
   char **ptr[ ] = {s+3, s+2, s+1, s};
   char ***p;
   p = ptr;
   ++p;
   printf("%s\n", (*(*p+1)+1)+2);
```

```
}
```

```
#include<stdio.h>
void main()
{
   char *s[ ]={"black", "white", "pink", "violet"};
   char **ptr[ ] = {s+1, s, s+3, s+2};
   char ***p;
   p = ptr;
   p+1;
   printf("%c \n", *(*(*++p+1))+3);
}
```

A function returning string:

- We can return a character using char data type.
- To return multiple characters (String) we use char* type.
- A function can return the string using char* return type.

```
#include<stdio.h>
char* read();
int main()
{
        char* name;
        name = read();
        printf("Hello %s \n", name);
        return 0;
}
char* read()
{
        char* name;
        printf("Enter your name :");
        gets(name);
        return name;
}
```