

Module 9

Pointers

9.1 Idea of Pointers

A Pointer in C language is a variable which holds the address of another variable of same data type. Pointers are used to access memory and manipulate the address.

Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language. Although pointers may appear a little confusing and complicated in the beginning, but trust me, once you understand the concept, you will be able to do so much more with C language.

Address in C

Whenever a variable is defined in C language, a memory location is assigned for it, in which its value will be stored. We can easily check this memory address, using the & symbol.

If var is the name of the variable, then &var will give its address.

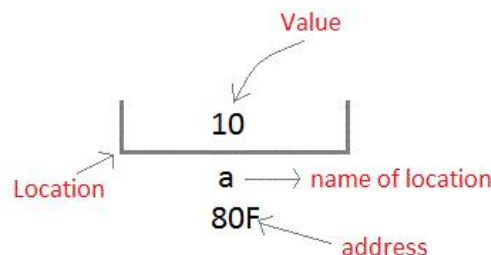
Example:

```
#include<stdio.h>
void main()
{
    int var = 7;
    printf("Value of the variable var is: %d\n", var);
    printf("Memory address of the variable var is: %x\n", &var);
}
```

Whenever a variable is declared in a program, system allocates a location i.e an address to that variable in the memory, to hold the assigned value. This location has its own address number, which we just saw above.

Let us assume that system has allocated memory location 80F for a variable a.

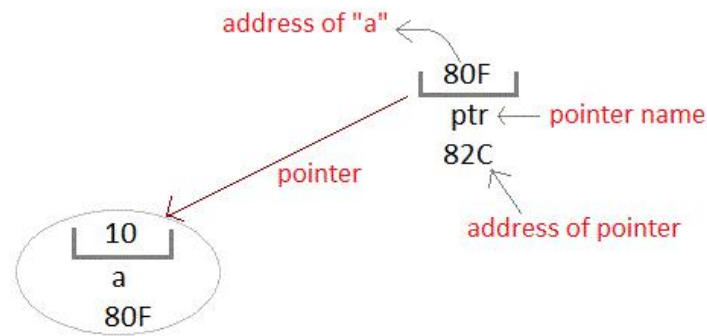
```
int a = 10;
```



We can access the value 10 either by using the variable name a or by using its address 80F.

The question is how we can access a variable using its address? Since the memory addresses are also just numbers, they can also be assigned to some other variable. The variables which are used to hold memory addresses are called Pointer variables.

A pointer variable is therefore nothing but a variable which holds an address of some other variable. And the value of a pointer variable gets stored in another memory location.



Benefits of using pointers

Below we have listed a few benefits of using pointers:

1. Pointers are more efficient in handling Arrays and Structures.
2. Pointers allow references to function and thereby helps in passing of function as arguments to other functions.
3. It reduces length of the program and its execution time as well.
4. It allows C language to support Dynamic Memory management.

9.2 Defining Pointers

General syntax of pointer declaration is,

```
datatype *pointer_name;
```

Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing. void type pointer works with all data types, but is not often used.

Here are a few examples:

```
int *ip      // pointer to integer variable
float *fp;   // pointer to float variable
double *dp;  // pointer to double variable
char *cp;    // pointer to char variable
```

Initialization of C Pointer variable

Pointer Initialization is the process of assigning address of a variable to a pointer variable. Pointer variable can only contain address of a variable of the same data type. In C language address operator & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
#include<stdio.h>
void main()
{
    int a = 10;
    int *ptr;      //pointer declaration
    ptr = &a;      //pointer initialization
}
```

Pointer variable always point to variables of same datatype. Let's have an example to showcase this:

```
#include<stdio.h>
void main()
{
    float a;
    int *ptr;
    ptr = &a;      // ERROR, type mismatch
}
```

If you are not sure about which variable's address to assign to a pointer variable while declaration, it is recommended to assign a NULL value to your pointer variable. A pointer which is assigned a NULL value is called a NULL pointer.

```
#include <stdio.h>
```

```

int main()
{
    int *ptr = NULL;
    return 0;
}

```

Using the pointer or Dereferencing of Pointer

Once a pointer has been assigned the address of a variable, to access the value of the variable, pointer is dereferenced, using the indirection operator or dereferencing operator *.

```
#include <stdio.h>
```

```

int main()
{
    int a, *p; // declaring the variable and pointer
    a = 10;
    p = &a;    // initializing the pointer

    printf("%d", *p); //this will print the value of 'a'
    printf("%d", *&a); //this will also print the value of 'a'
    printf("%u", &a); //this will print the address of 'a'
    printf("%u", p); //this will also print the address of 'a'
    printf("%u", &p); //this will print the address of 'p'
    return 0;
}

```

Points to remember while using pointers

1. While declaring/initializing the pointer variable, * indicates that the variable is a pointer.
2. The address of any variable is given by preceding the variable name with Ampersand &.
3. The pointer variable stores the address of a variable. The declaration int *a doesn't mean that a is going to contain an integer value. It means that a is going to contain the address of a variable storing integer value.
4. To access the value of a certain address stored by a pointer variable, * is used. Here, the * can be read as 'value at'.

Example

```
#include <stdio.h>
```

```

int main()
{
    int i = 10; // normal integer variable storing value 10
    int *a;    // since '*' is used, hence its a pointer variable
    /*
        '&' returns the address of the variable 'i'
        which is stored in the pointer variable 'a'
    */
    a = &i;
    /*
        below, address of variable 'i', which is stored
        by a pointer variable 'a' is displayed
    */
    printf("Address of variable i is %u\n", a);
    /*
        below, '*a' is read as 'value at a'
        which is 10
    */
    printf("Value at the address, which is stored by pointer variable a is %d\n", *a);
    return 0;
}

```

9.3 Use of Pointers in Self-referential Structures

Structures can have members which are of the type the same structure itself in which they are included, This is possible with pointers and the phenomenon is called as self referential structures.

A self referential structure is a structure which includes a member as pointer to the present structure type.

The general format of self referential structure is

```
struct parent
{
  memeber1;
  memeber2;
  -----;
  -----;
  struct parent *name;
};
```

The structure of type parent is contains a member, which is pointing to another structure of the same type i.e. parent type and name refers to the name of the pointer variable.

Here, name is a pointer which points to a structure type and is also an element of the same structure.

Example

```
struct element
{
  char name{20}; int num;
  struct element * value;
}
```

Element is of structure type variable. This structure contains three members

- a 20 elements character array called name
- An integer element called num
- a pointer to another structure which is same type called value. Hence it is self referential structure.

These structure are mainly used in applications where there is need to arrange data in ordered manner.

9.4 Notion of linked List

A linked list is a dynamic data structure where each element (called a node) is made up of two items - the data and a reference (or pointer) which points to the next node. A linked list is a collection of nodes where each node is connected to the next node through a pointer. The first node is called a head and if the list is empty then the value of head is NULL.

A Simple Linked List



For a real-world analogy of linked list, you can think of conga line, a special kind of dance in which people line up behind each other with hands on shoulders of the person in front. Each dancer represents a data element while their hands serve as the pointers or links to the next element.

Advantages of Linked Lists

1. The dynamic nature of linked list comes with a number of advantages.
2. In contrast to arrays, which have pre-defined or fixed length, linked lists have a dynamic length which can be increased or decreased at runtime.
3. Insertion and deletion operations in the linked list are much faster in comparison to other data structure such as the queue, stack, and arrays.
4. Consequently, it is often better to consider using a list when the exact volume and quantity is not known ahead of time and cannot be made fixed. For instance, a programmer designing a school management system cannot determine how many students will enroll in the school. Therefore, it is most efficient to choose an ordered list data structure over arrays.

Linked list as Self-referencing Structure

As self-referential structure means that at least one member of the structure is a pointer to the structure of its own type. An implementation of self-referential structure is as follows:

```
struct Node {
    int data;
    struct Node *next;
}
```

In the above code, the structure node contains data element data as well as pointer next which points to the structure of the same type. The (*) indicates a pointer definition and it points to the address of the next node of the linked list. As the linked list is traversed using the next pointer, the value of the pointer in the last node will be NULL. The self-referential structure is the reason why a linked list is called a dynamic data structure and can be expanded and pruned at runtime.

Ordered List vs Ordered Array

Suppose we have an ordered array arr[] having integer values.

```
arr[] = [50, 100, 120, 150, 200];
```

If we have to insert a new value 70 into the array, then we have to move all elements after 50 to maintain the ordered array. Similarly, if we have to delete a value 100 from the array, we have to move all the values after 100. So, insert and delete operations are expensive in ordered arrays. If we maintain an ordered list, the insert and delete operations are faster and more efficient due to the use of pointers.

9.5 Pointer to Pointer

Pointers are used to store the address of other variables of similar datatype. But if you want to store the address of a pointer variable, then you again need a pointer to store it. Thus, when one pointer variable stores the address of another pointer variable, it is known as Pointer to Pointer variable or Double Pointer.

Syntax:

```
int **p1;
```

Here, we have used two indirection operator(*) which stores and points to the address of a pointer variable i.e, int *. If we want to store the address of this (double pointer) variable p1, then the syntax would become:

```
int ***p2
```

Simple program to represent Pointer to a Pointer

```
#include <stdio.h>
int main() {
    int a = 10;
    int *p1;          //this can store the address of variable a
    int **p2;
    /*
        this can store the address of pointer variable p1 only.
        It cannot store the address of variable 'a'
    */
    p1 = &a;
    p2 = &p1;
    printf("Address of a = %u\n", &a);
    printf("Address of p1 = %u\n", &p1);
    printf("Address of p2 = %u\n", &p2);
    // below print statement will give the address of 'a'
    printf("Value at the address stored by p2 = %u\n", *p2);
    printf("Value at the address stored by p1 = %d\n", *p1);
    printf("Value of **p2 = %d\n", **p2); //read this *(*p2)
    /*
```

```

        This is not allowed, it will give a compile time error-
        p2 = &a;
        printf("%u", p2);
    */
    return 0;
}

```

Output:

```

Address of a = 2686724
Address of p1 = 2686728
Address of p2 = 2686732
Value at the address stored by p2 = 2686724
Value at the address stored by p1 = 10
Value of **p2 = 10

```

Explanation of the above program



- p1 pointer variable can only hold the address of the variable a (i.e Number of indirection operator(*)-1 variable). Similarly, p2 variable can only hold the address of variable p1. It cannot hold the address of variable a.
- *p2 gives us the value at an address stored by the p2 pointer. p2 stores the address of p1 pointer and value at the address of p1 is the address of variable a. Thus, *p2 prints address of a.
- **p2 can be read as *(*p2). Hence, it gives us the value stored at the address *p2. From above statement, you know *p2 means the address of variable a. Hence, the value at the address *p2 is 10. Thus, **p2 prints 10.

9.6 Pointer to Array

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e address of the first element of the array is also allocated by the compiler.

Suppose we declare an array arr,

```
int arr[5] = { 1, 2, 3, 4, 5 };
```

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

element	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
Address	1000	1002	1004	1006	1008

Here variable arr will give the base address, which is a constant pointer pointing to the first element of the array, arr[0]. Hence arr contains the address of arr[0] i.e 1000. In short, arr has two purpose - it is the name of the array and it acts as a pointer pointing towards the first element in the array.

arr is equal to &arr[0] by default

We can also declare a pointer of type int to point to the array arr.

```

int *p;
p = arr;

```

```
// or,
p = &arr[0]; //both the statements are equivalent.
```

Now we can access every element of the array arr using p++ to move from one element to another.

As studied above, we can use a pointer to point to an array, and then we can use that pointer to access the array elements. Lets have an example,

```
#include <stdio.h>
int main()
{
    int i;
    int a[5] = {1, 2, 3, 4, 5};
    int *p = a; // same as int*p = &a[0]
    for (i = 0; i < 5; i++)
    {
        printf("%d", *p);
        p++;
    }

    return 0;
}
```

In the above program, the pointer *p will print all the values stored in the array one by one. We can also use the Base address (a in above case) to act as a pointer and print all the values.

Replacing the **printf("%d", *p);** statement of above example, with below mentioned statements. Lets see what will be the result.

printf("%d", a[i]); —→ **prints the array, by incrementing index**

printf("%d", i[a]); —→ **this will also print elements of array**

printf("%d", a+i); —→ **This will print address of all the array elements**

printf("%d", *(a+i)); —→ **Will print value of array element.**

printf("%d", *a); —→ **will print value of a[0] only**

a++; —→ **Compile time error, we cannot change base address of the array.**

The generalized form for using pointer with an array,

```
*(a+i)
is same as:
a[i]
```

Pointer to Multidimensional Array

A multidimensional array is of form, a[i][j]. Lets see how we can make a pointer point to such an array. As we know now, name of the array gives its base address. In a[i][j], a will give the base address of this array, even a + 0 + 0 will also give the base address, that is the address of a[0][0] element.

Here is the generalized form for using pointer with multidimensional arrays.

```
*(*(a + i) + j)
which is same as,
a[i][j]
```

9.7 Pointer to Strings

Pointer can also be used to create strings. Pointer variables of char type are treated as string.

```
char *str = "Hello";
```

The above code creates a string and stores its address in the pointer variable str. The pointer str now points to the first character of the string "Hello". Another important thing to note here is that the string created using char pointer can be assigned a value at runtime.

```
char *str;  
str = "hello";    //this is Legal
```

The content of the string can be printed using printf() and puts().

```
printf("%s", str);  
puts(str);
```

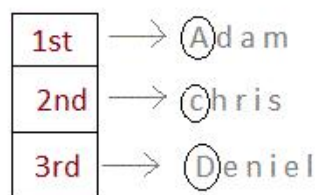
Notice that str is pointer to the string, it is also name of the string. Therefore we do not need to use indirection operator *.

9.8 Array of Pointer

We can also have array of pointers. Pointers are very helpful in handling character array with rows of varying length.

```
char *name[3] = {  
    "Adam",  
    "chris",  
    "Deniel"  
};  
//Now lets see same array without using pointer  
char name[3][20] = {  
    "Adam",  
    "chris",  
    "Deniel"  
};
```

Using Pointer



char* name[3]

Only 3 locations for pointers, which will point to the first character of their respective strings.

Without Pointer



char name[3][20]

extends till 20 memory locations

In the second approach memory wastage is more, hence it is preferred to use pointer in such cases. When we say memory wastage, it doesn't mean that the strings will start occupying less space, no, characters will take the same space, but when we define array of characters, a contiguous memory space is located equal to the maximum size of the array, which is a wastage, which can be avoided if we use pointers instead.

9.9 Pointer to Function

Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as call by reference. When a function is called by reference any change made to the reference variable will effect the original variable.

Example: Swapping two numbers using Pointer

```
#include <stdio.h>
void swap(int *a, int *b);
int main()
{
    int m = 10, n = 20;
    printf("m = %d\n", m);
    printf("n = %d\n\n", n);
    swap(&m, &n);    //passing address of m and n to the swap function
    printf("After Swapping:\n\n");
    printf("m = %d\n", m);
    printf("n = %d", n);
    return 0;
}
/*
    pointer 'a' and 'b' holds and
    points to the address of 'm' and 'n'
*/
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Output:

```
m = 10
n = 20
After Swapping:
m = 20
n = 10
```

Functions returning Pointer variables

A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only inside the function. Hence if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends.

```
#include <stdio.h>
int* larger(int*, int*);
void main()
{
    int a = 15;
    int b = 92;
    int *p;
    p = larger(&a, &b);
    printf("%d is larger", *p);
}
int* larger(int *x, int *y)
{
    if(*x > *y)
        return x;
    else
```

```

        return y;
    }

```

Output: 92 is larger

Safe ways to return a valid Pointer.

1. Either use argument with functions. Because argument passed to the functions are declared inside the calling function, hence they will live outside the function as well.
2. Or, use static local variables inside the function and return them. As static variables have a lifetime until the main() function exits, therefore they will be available throughout the program.

Pointer to functions

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. A pointer to a function is declared as follows,

type (*pointer-name)(parameter);

Example :

```

int (*sum)();    //legal declaration of pointer to function
int *sum();      //This is not a declaration of pointer to function

```

A function pointer can point to a specific function when it is assigned the name of that function.

```

int sum(int, int);
int (*s)(int, int);
s = sum;

```

Here s is a pointer to a function sum. Now sum can be called using function pointer s along with providing the required argument values.

```

s (10, 20);

```

Example of Pointer to Function

```

#include <stdio.h>

```

```

int sum(int x, int y)
{
    return x+y;
}

int main( )
{
    int (*fp)(int, int);
    fp = sum;
    int s = fp(10, 15);
    printf("Sum is %d", s);

    return 0;
}

```

Output: 25

Complicated Function Pointer example

You will find a lot of complex function pointer examples around, lets see one such example and try to understand it.

```

void *(*foo) (int*);

```

It appears complex but it is very simple. In this case (*foo) is a pointer to the function, whose argument is of int* type and return type is void*.

9.10 Pointer to Structure

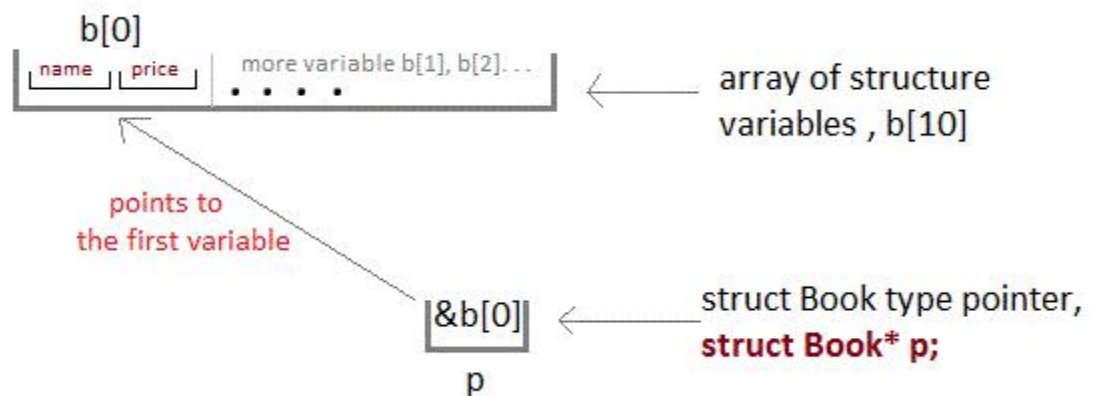
Like we have array of integers, array of pointers etc, we can also have array of structure variables. And to use the array of structure variables efficiently, we use pointers of structure type. We can also have pointer to a single structure variable, but it is mostly used when we are dealing with array of structure variables.

```
#include <stdio.h>
struct Book
{
    char name[10];
    int price;
}

int main()
{
    struct Book a;      //Single structure variable
    struct Book* ptr;   //Pointer of Structure type
    ptr = &a;

    struct Book b[10];  //Array of structure variables
    struct Book* p;      //Pointer of Structure type
    p = &b;

    return 0;
}
```



Accessing Structure Members with Pointer

To access members of structure using the structure variable, we used the dot `.` operator. But when we have a pointer of structure type, we use arrow `->` to access structure members.

```
#include <stdio.h>
struct my_structure {
    char name[20];
    int number;
    int rank;
};

int main()
{
    struct my_structure variable = {"StudyTonight", 35, 1};

    struct my_structure *ptr;
    ptr = &variable;

    printf("NAME: %s\n", ptr->name);
}
```

```
    printf("NUMBER: %d\n", ptr->number);  
    printf("RANK: %d", ptr->rank);  
  
    return 0;  
}
```

NAME: StudyTonight
NUMBER: 35
RANK: 1