**Leaning Objective:**

In this Module you will be learning the following:

- Pointers

**Introduction:**

Pointers in C language is a variable that stores/points the address of another variable. A Pointer in C is used to allocate memory dynamically i.e. at run time. Pointers are used in C program to access the memory and manipulate the address.

**Material:**

Pointers are powerful features of C and (C++) programming that differentiates it from other popular programming languages like: Java and Python.

The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.

**Pointer Syntax:** data_type *var_name;

**Example:** int *p;  char *p;

Where, * is used to denote that "p" is pointer variable and not a normal variable.

**Key points to remember about pointers in c:**

- Normal variable stores the value whereas pointer variable stores the address of the variable.
- The content of the C pointer always be a whole number i.e. address.
- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If a pointer in C is assigned to NULL, it means it is pointing to nothing.
- Two pointers can be subtracted to know how many elements are available between these two pointers.
- But, Pointer addition, multiplication, division are not allowed.
- The size of any pointer is 2 byte (for 16 bit compiler).

**Address in C**

Before you get into the concept of pointers, let's first get familiar with address in C. If you have a variable var in your program, &var will give you its address in the memory, where & is commonly called the reference operator.

You must have seen this notation while using scanf() function. It was used in the function to store the user inputted value in the address of var.

scanf("%d", &var);

**Reference operator (&) and Dereference operator (*)**

As discussed, & is called reference operator. It gives you the address of a variable.

Likewise, there is another operator that gets you the value from the address, it is called a dereference operator (*).

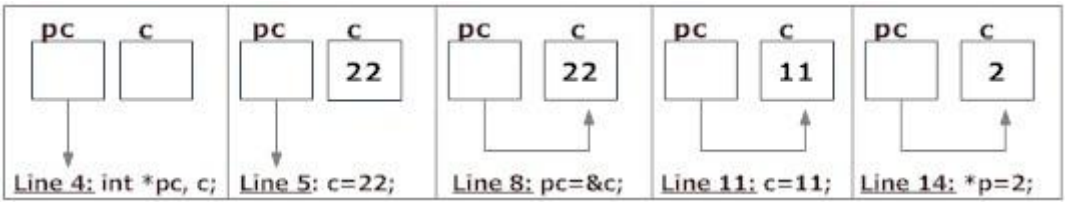
**Note:** The * sign when declaring a pointer is not a dereference operator. It is just a similar notation that creates a pointer.

**Example program for pointers in c:**

```
/* Source code to demonstrate, handling of pointers in C program */
#include <stdio.h>
int main(){
  int* pc;
  int c;
  c=22;
  printf("Address of c:%u\n",&c);
  printf("Value of c:%d\n\n",c);
  pc=&c;
  printf("Address of pointer pc:%u\n",pc);
  printf("Content of pointer pc:%d\n\n",*pc);
  c=11;
  printf("Address of pointer pc:%u\n",pc);
  printf("Content of pointer pc:%d\n\n",*pc);
  *pc=2;
  printf("Address of c:%u\n",&c);
  printf("Value of c:%d\n\n",c);
  return 0;
}
```

**OUTPUT:**

```
Address of c: 2686784
Value of c: 22
Address of pointer pc: 2686784
Content of pointer pc: 22
Address of pointer pc: 2686784
Content of pointer pc: 11
Address of c: 2686784
Value of c: 2
```




**Explanation of program and figure**

1. int* pc; creates a pointer pc and int c; creates a normal variable c. Since pc and c are both not initialized, pointer pc points to either no address or a random address. Likewise, variable c is assigned an address but contains a random/garbage value.
2. c=22; assigns 22 to the variable c, i.e., 22 is stored in the memory location of variable c. Note that, when printing &c (address of c), we use %u rather than %d since address is usually expressed as an unsigned integer (always positive).
3. pc=&c; assigns the address of variable to c to the pointer pc. When printing, you see value of pc is the same as the address of c and the content of pc (*pc) is 22 as well.
4. c=11; assigns 11 to variable c. We assign a new value to c to see its effect on pointer pc.
5. Since, pointer pc points to the same address as c, value pointed by pointer pc is 11 as well. Printing the address and content of pc shows the updated content as 11.
6. *pc=2; changes the contents of the memory location pointed by pointer pc to 2. Since the address of pointer pc is same as address of c, value of c also changes to 2.

**Common mistakes when working with pointers**

Suppose, you want pointer pc to point to the address of c. Then,

```
int c, *pc;
// Wrong! pc is address whereas, c is not an address.
pc = c;
// Wrong! *pc is the value pointed by address whereas, %amp;c is an address.
*pc = &c;
// Correct! pc is an address and, %amp;pc is also an address.
pc = &c;
// Correct! *pc is the value pointed by address and, c is also a value.
*pc = c;
```

In both cases, pointer pc is not pointing to the address of c.

**Pointers and Arrays**

Arrays are closely related to pointers in C programming but the important difference between them is that, a pointer variable takes different addresses as value whereas, in case of array it is fixed.

This can be demonstrated by an example:

```
#include <stdio.h>
void main(){
  char charArr[4];
  int i;
  for(i = 0; i < 4; ++i)   {
    printf("Address of charArr[%d] = %u\n", i, &charArr[i]);
  }
}
```

**When you run the program, the output will be:**

```
Address of charArr[0] = 28ff44
Address of charArr[1] = 28ff45
Address of charArr[2] = 28ff46
Address of charArr[3] = 28ff47
```

**Note:** You may get different address of an array.

Notice, that there is an equal difference (difference of 1 byte) between any two consecutive elements of array charArr. But, since pointers just point at the location of another variable, it can store any address.

**Relation between Arrays and Pointers**

Consider an array: int arr[4];

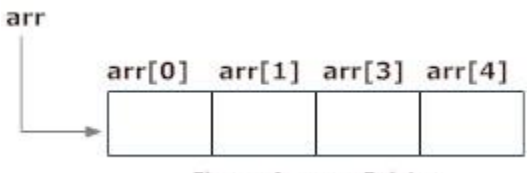



Figure: Array as Pointer

In C programming, name of the array always points to address of the first element of an array. In the above example, arr and &arr[0] points to the address of the first element.

**&arr[0] is equivalent to arr**

Since, the addresses of both are the same, the values of arr and &arr[0] are also the same.

arr[0] is equivalent to *arr (value of an address of the pointer). Similarly,

&arr[1] is equivalent to (arr + 1) AND, arr[1] is equivalent to *(arr + 1).

&arr[2] is equivalent to (arr + 2) AND, arr[2] is equivalent to *(arr + 2).

&arr[3] is equivalent to (arr + 3) AND, arr[3] is equivalent to *(arr + 3).

.

.

&arr[i] is equivalent to (arr + i) AND, arr[i] is equivalent to *(arr + i).

In C, you can declare an array and can use pointer to alter the data of an array.

**Example: Program to find the sum of six numbers with arrays and pointers**

```
#include <stdio.h>
int main(){
  int i, classes[6],sum = 0;
  printf("Enter 6 numbers:\n");
  for(i = 0; i < 6; ++i)  {
     // (classes + i) is equivalent to &classes[i]
     scanf("%d",(classes + i));
     // *(classes + i) is equivalent to classes[i]
     sum += *(classes + i);
  }
  printf("Sum = %d", sum);
  return 0;
}
```

**Output**

```
Enter 6 numbers:    2       3       4       5       3       4
Sum = 21
```

**C Call by Reference: Using pointers**

When a pointer is passed as an argument to a function, address of the memory location is passed instead of the value. This is because, pointer stores the location of the memory, and not the value.

**Example of Pointer And Functions**

Program to swap two number using call by reference.

```
/* C Program to swap two numbers using pointers and function. */
#include <stdio.h>
void swap(int *n1, int *n2);
void main(){
   int num1 = 5, num2 = 10;
   // address of num1 and num2 is passed to the swap function
   swap( &num1, &num2);
   printf("Number1 = %d\n", num1);
   printf("Number2 = %d", num2);
}
```

```
void swap(int * n1, int * n2){
   // pointer n1 and n2 points to the address of num1 and num2 respectively
   int temp;
   temp = *n1;
   *n1 = *n2;
   *n2 = temp;
}
```

**Output**

```
Number1 = 10
Number2 = 5
```

The address of memory location num1 and num2 are passed to the function swap and the pointers *n1 and *n2 accept those values. So, now the pointer n1 and n2 points to the address of num1 and num2 respectively. When, the value of pointers are changed, the value in the pointed memory location also changes correspondingly. Hence, changes made to *n1 and *n2 are reflected in num1 and num2 in the main function.

This technique is known as Call by Reference in C programming.

### C Dynamic Memory Allocation

In C, the exact size of array is unknown until compile time, i.e., the time when a compiler compiles your code into a computer understandable language. So, sometimes the size of the array can be insufficient or more than required. Dynamic memory allocation allows your program to obtain more memory space while running, or to release it if it's not required. In simple terms, Dynamic memory allocation allows you to manually handle memory space for your program. Although, C language inherently does not have any technique to allocate memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

| Function | Use of Function |
|---|---|
| malloc() | Allocates requested size of bytes and returns a pointer first byte of allocated space |
| calloc() | Allocates space for an array elements, initializes to zero and then returns a pointer to memory |
| free() | deallocate the previously allocated space |
| realloc() | Change the size of previously allocated space |

### C malloc()

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

**Syntax of malloc()**

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

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

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

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

**C calloc()**

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

**Syntax of calloc()**

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

This statement will allocate contiguous space in memory for an array of n elements. For example:

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

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

**C free()**

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

**syntax of free()**

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

**Example #1: Using C malloc() and free()**

**Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.**

```
#include <stdio.h>
#include <stdlib.h>
int  main(){
   int num, i, *ptr, sum = 0;
   printf("Enter number of elements: ");
   scanf("%d", &num);
   ptr = (int*) malloc(num * sizeof(int));  //memory allocated using malloc
   if(ptr == NULL) {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i = 0; i < num; ++i)   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }
   printf("Sum = %d", sum);
   free(ptr);
   return 0;
}
```

**Example #2: Using C calloc() and free()**

**Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.**

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

**C realloc()**

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

**Syntax of realloc()**

```
ptr = realloc(ptr, newsize);
```

Here, ptr is reallocated with size of newsize.

**Example #3: Using realloc()**

```
#include <stdio.h>
#include <stdlib.h>
int main(){
   int *ptr, i , n1, n2;
   printf("Enter size of array: ");
   scanf("%d", &n1);
   ptr = (int*) malloc(n1 * sizeof(int));
   printf("Address of previously allocated memory: ");
   for(i = 0; i < n1; ++i)
       printf("%u\t",ptr + i);
```

```
    printf("\nEnter new size of array: ");
    scanf("%d", &n2);
    ptr = realloc(ptr, n2);
    for(i = 0; i < n2; ++i)
        printf("%u\t", ptr + i);
    return 0;
}
```

**Problem Set**

1. Write a C program to create, initialize, assign and access a pointer variable.
2. Write a program in C to demonstrate the use of &(address of) and *(value at address) operator.
3. Write a C program which declares int, real, character pointer variables. Assign values to declared pointer variables and print the value of pointer variable pointing to and the value of the address holding by pointer variable.
4. Write a C Program to print size of different types of pointer variables.
5. Write a C Program to Calculate Average of an Array Using Pointer
6. Write a C Program to print a string using pointer.
7. Write a C Program to count vowels and consonants in a string using pointer.
8. Write a C Write a program in C to sort an array using Pointer.
9. Write a C Program to perform Stack Operations Using Pointer!
10. Write a C Program to read integers into an array and reversing them using pointers
11. Write a C Program to Add Two Matrix Using Multi-dimensional Arrays
12. Write a C Program to Multiply to Matrix Using Multi-dimensional Arrays
13. Write a C Program to Find Transpose of a Matrix
14. Write a C Program to Multiply two Matrices by Passing Matrix to Function
15. Write a C Program to Sort Elements of an Array
16. Write a C Program to Find Largest Number Using Dynamic Memory Allocation