TCS201: Programming and Problem Solving
UNIT-2
Pointers & Structures

POINTER

A pointer is a special variable that holds the address of another variable i.e. direct address of the memory location. Like any variable or constant, it must be declared before using it to store any variable address.

DECLARING A POINTER VARIABLE

The syntax is shown below:

data_type *ptr_var_name;

Here, **data_type** is the pointer's base type; it must be a valid C data type and **ptr_var_name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk being used indicates a variable as a pointer.

For ex:

```
int     *ip;    /* pointer to an integer */
double  *dp;    /* pointer to a double */
float   *fp;    /* pointer to a float */
char    *ch;    /* pointer to a character */
```

The actual value that all pointers, whether integer, float or character points to contains the same long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Steps to access pointers:

1) Declare a variable of basic type ex: int var;
2) Declare a pointer-variable of the same type as in step one ex: int *ptr;
3) Initialize the pointer-variable ex: ptr=&var;
4) Access the value using the pointer-variable ex: printf("%d",*ptr);

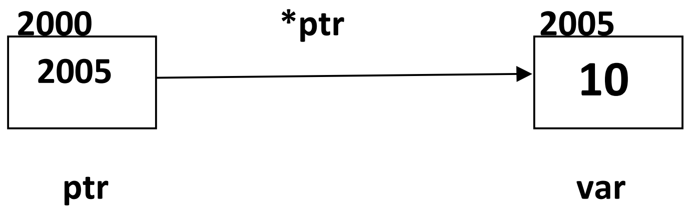
For ex:
Program to access the values using pointers:

```
#include<stdio.h>
void main()
{       int var=10;
        char ch='d';
        int *ptr;
        int *cptr;
        ptr=&var;
        cptr=&ch;
        printf("Address of var: %u \n", ptr);
        printf("Address of ch: %u \n\n", cptr);
        printf("Value of var thru pointer: %d \n", *ptr);
        printf("Value of ch thru pointer: %c", *cptr);
return 0;
}
```

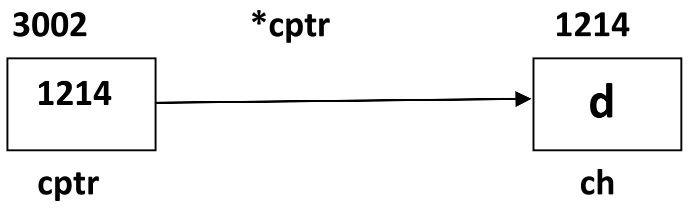
Output*:*

```
Address of var:                2703882588
Address of ch:                 2703882587
Value of var thru pointer:     10
Value of ch thru pointer:      d
```

In the above program ptr is a **pointer** of **int type** and the statement ptr=&var stores the address of the variable **var** of integer type i.e 2005 in the above case.
Similarly, cptr is a **pointer** of **char type** and the statement ptr=&ch stores the address of the variable **ch** of char type i.e 1214 in the above case. Diagrammatic representation of pointer used in the above program:



*ptr also called dereferencing a pointer, points to the value i.e 10 in the above case.



Similarly,***cptr** is called dereferencing the pointer, ie points to the actual value **d** in the above case.

Ex. 1) Any variable that is declared is stored in a memory and every memory-location has an address, and the address can be accessed using the ampersand (&) operator. Consider the following example, that prints the address of the variables defined:

```
#include <stdio.h>
int main ()
{  int  var = 20;   /* actual variable declaration */
   int  *ptr;       /* pointer variable declaration */

   ptr = &var;  /* store address of var in pointer variable*/
   printf("Address of the var variable: %u\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ptr variable: %u\n", ptr );

   /* accessing the value using the pointer */
   printf("Value of *ptr variable: %d\n", *ptr );
   return 0;
}
```

Output:

```
Address of the var variable:      3856272196
Address stored in ptr variable:   3856272196
Value of *ptr variable:           20
```

Ex. 2)

```
#include <stdio.h>
int main ()
{   int var1[2],*p;
   char ch[2],*c;
   float *f[2];
   p=&var1[0];
   c=&ch[0];
   printf("Address: %u %u %u %u\n", p,(p+1),c,(c+1));
   printf("Address of var2 variable: %u %u\n", &var1,&ch[1]  );

   return 0;
}
```

Ex. 3)

```
#include<stdio.h>
void main()
{
        int var=10;
        char ch='d';
        printf("Address of variable var: %u \n", &var);
        printf("Address of variable ch:   %u   ", &ch);
        return 0;
}
```

Output*:*

```
        Address of variable var: 3185445644
        Address of variable ch:     3185445643
```

Important points to remember about pointers:

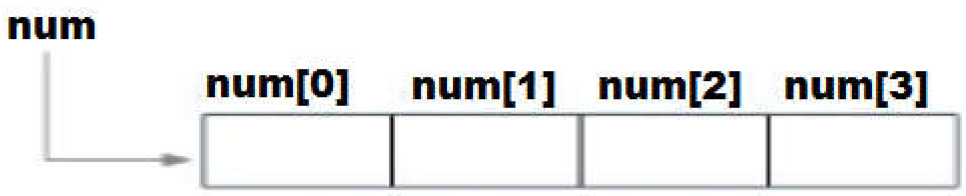
- Normal variable stores the value whereas pointer variable stores the address of the variable.
- The content of a pointer always be a whole number i.e. address.
- Pointer can be initialized to null, i.e. int *p = NULL.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If a pointer 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.
- Pointer addition, multiplication, division are not allowed.
- The size of any pointer is 2 byte (for 16 bit compiler).

POINTERS AND ARRAYS

Consider an array:

```
int num[4];
```

- The above code can be pictorially represented as shown below:



- The name of the array always points to the first element of an array.
- Here, address of first element of the array is &num[0].
- Also, **num** represents the address of the pointer where it is pointing. Hence, &num[0] is equivalent to **num** and is the **first address of the array**.
- Also, value inside the address **&num[0]** and address **num** are equal. Value in address &num[0] is num[0] and value in **address num** is *num. Hence, num[0] is equivalent to *num.

Similarly,
&num [1] is equivalent to (num+1) AND, num[1] is equivalent to *(num+1).
&num [2] is equivalent to (num+2) AND, num[2] is equivalent to *(num+2).
&num [3] is equivalent to (num+3) AND, num[3] is equivalent to *(num+3).
.
.
&num[i] is equivalent to (num+i) AND, num[i] is equivalent to *(num+i).

- Pointers can be used to alter the data of an array.
- Pointers can be used alternatively in place of arrays.

- Example:

Ex. 1) Program to access elements of an array using pointer.

```
#include<stdio.h>
void main()
{
        int data[5], i;

        printf("Enter the elements: ");
        for(i=0;i<5;++i)
                scanf("%d", (data+i) );

        printf("You entered the following elements:");

        for(i=0;i<5;++i)
                printf("%d ",*(data+i) );

}
```

Output:

```
Enter the elements: 1 2 3 5 4
```

```
You entered the following elements: 1 2 3 5 4
```

Ex. 2) Program to find sum of all the elements of an array using array as pointers.

```
#include<stdio.h>
void main()
{
        int i, num[10], n, sum=0;

        printf("Enter the size of the array:");
        scanf("%d", &n);
        printf("\nEnter the elements into the array: ");
        for(i=0;i<n;i++)
                scanf("%d ",num+i);

        for(i=0;i<n;i++)
                sum+=*(num+i);

        printf("\nSum is %d ",sum);
}
```

Output:

```
Enter the size of the array: 5
Enter the elements: 2 4 6 8 10
Sum is 30
```

Ex. 3) Program to find sum of all the elements of an array using another pointer.

```
#include<stdio.h>
void main()
{
        int i, num[10], n, sum=0;
        int *ptr;
        ptr=&num;   // or ptr=num or ptr=&num[0]; assigns the base address of num to
                    //the pointer ptr
        printf("Enter the size of the array:");
        scanf("%d", &n);
        printf("\nEnter the elements into the array: ");
        for(i=0;i<n;i++)
                scanf("%d ",num+i);

        for(i=0;i<n;i++)
        {       sum+=*ptr;
```

```
                ptr++;
        }
        printf("\nSum is %d ",sum);
}
```

Output:

```
Enter the size of the array: 5
Enter the elements: 2 4 6 8 10
Sum is 30
```

POINTER ARITHMETIC

- Pointer is an address, which is a numeric value. Therefore, one can perform arithmetic operations on a pointer just like on any other numeric value.
- There are four arithmetic operators that can be used on pointers: ++, --, +, and –
- To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000.
- Assuming 16-bit integers, let us perform the following arithmetic operation on the pointer: ptr++
- Now, after the above operation, the **ptr** will point to the location 1002 because each time **ptr** is incremented by 16-bits in this case, i.e it will point to the next integer location that is 2 bytes next to the current location.

Incrementing a Pointer

- Pointers are preferred in a program instead of an array because the variable pointer can be incremented, unlike the **array** name which cannot be incremented because it is a **constant pointer**.

- Example: Program to increment the variable pointer to access each succeeding element of the array.

```
#include<stdio.h>
void main()
{
        int list[] = {111,222,333};
        int i, *ptr;
        ptr = list;
        for ( i = 0; i < 3; i++)
        {       printf("Address of list[%d] = %u \n", i, ptr );
                printf("Value of list[%d] = %d \n", i, *ptr );
                ptr++; //move to the next location
        }
}
```

Output:

```
Address of list[0] = 2000
Value of list[0] = 111
Address of list[1] = 2002
Value of list[1] = 222
Address of list[2] = 2004
Value of list[2] = 333
```

Following is another version of the earlier example using pointers.

Ex. 3) Program to find sum of all the elements of an array using pointers.

```
#include<stdio.h>
void main()
{
        int i, num[10], n, sum=0;
        int *ptr;

        printf("Enter the size of the array:");
        scanf("%d", &n);
        ptr=num;
        printf("Enter the elements into the array: ");
        for(i=0;i<n;i++)
        {   scanf("%d",ptr);
            ptr++;
        }
        ptr=num;
        for(i=0;i<n;i++)
        {   sum+=*ptr;
            ptr++;
        }
        printf("Sum is %d ",sum);
}
```

Output:

```
Enter the size of the array: 5
Enter the elements into the array: 11 22 33 44 55
Sum is 165
```

Ex. 3) Program to display the array in reverse using pointers.

```
void main()
{
        int i, num[10], n;
        int *ptr;

        printf("Enter the size of the array:");
        scanf("%d", &n);

        ptr=num;   //assigns the base address of num to the pointer ptr

        printf("Enter the elements into the array: ");
        for(i=0;i<n;i++)
                scanf("%d",&num[i]);

        ptr=&num[4];

        printf("Elements in the reverse order: ");
        for(i=n;i>0;i--)
        {   printf("\n %d",*ptr);
            ptr--;
        }
}
```

Output:

```
Enter the size of the array: 5
Enter the elements into the array: 11 22 33 44 55
Elements in the reverse order        : 55 44 33 22 11
```

POINTERS AND FUNCTIONS

When, argument is passed using pointer, address of the memory-location is passed instead of value.

- Example: Program to swap two numbers using **call by reference**.

```
#include<stdio.h>
void swap(int *p1, int *p2 )
{       // pointer p1 and p2 points to the address of num1 and num2 respectively
        int temp;
        temp=*p1;
        *p1=*p2;
        *p2=temp;
}
void main()
{
        int num1=27,num2=300;
        printf("Before swapping num1= %d and num2=%d\n",num1,num2);
        swap(&num1, &num2); //address of num1 & num2 is passed to swap function
        printf("After swapping num1= %d and num2=%d\n",num1,num2);
}
```

Output:
Before swapping num1= 29 and num2= 300
After swapping num1= 300 and num2= 29

Table-1

Variable Name	Address	Value	Declared in
num1	2000	29	**main()**
num2	2004	300	

Table-2

Variable Name	Address	Value	Declared in
p1	5000	2000	**swap()**
P2	5004	2004	

Table-3

Variable Name	Address	Value	Declared in
num1	2000	300	**main()**
num2	2004	29	

Explanation

- The address of memory-locations ***num1*** and ***num2*** as shown in Table-1 are 2000 and 2004 respectively declared in the main() program.
- When the function ***swap(&num1,&num2)*** is called the address of num1 and num2 is passed i.e. 2000 and 2004 respt to the function ***swap***.
- The definition of the function ***swap*** shows two pointers ***p1** and ***p2** declared stores the address of ***num1*** and ***num2*** passed thru ***swap***.
- While the compiler allocates memory storage separately to the pointers ***p1** and ***p2** as shown in the Table-2.
- When, the value of pointer is changed by the following statement as shown in Table-3:

```
temp=*p1;   // copies the content of the memory 2000 (p1) ie 29 into temp
*p1=*p2;    // content of the memory 2000 (p1) is overwritten by the
            // value 300 stored in the address 2004 (p2)
*p2=temp; //content of the memory 2004 (p2) is over written by the value
          //29 stored in the variable temp
```

- The values in memory-location of ***num1*** and ***num2*** are changed respectively by the above statements by ***swap*** without the knowledge of the ***main*** function.
- Hence, changes made to ***p1** and ***p2** also gets reflected in ***num1*** and ***num2*** in the main function as shown in Table-3.
- This method of parameter passing technique is known as **call by reference**.

Example-1: Write a C program to reverse a string using a pointer and a recursive function.

```
#include <stdio.h>
void reverse(char *str,int i)
{
        if(*str=='\0')
                return;
        reverse(str+i,i++);
        printf("%c",*str);
}
int main()
{  char *str = "Hello World";
   reverse(str,0);
   printf("\n\n");
   return 0;
}
```

Output:

```
dlroW olleH
```

Example-2: Write a C program to find a string accepted from the user is palindrome or not using a pointer.

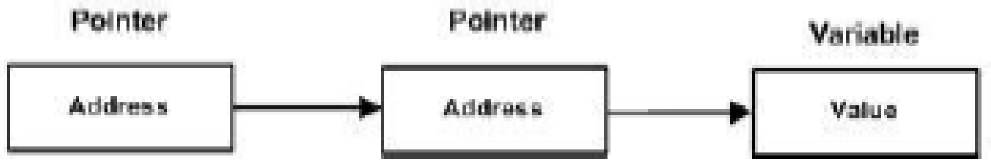
```
#include <stdio.h>
#include <string.h>
int Ispalindrome(char *ptr)
{       int i,len=strlen(ptr);
        int j=len-1;
        for(i=0;i<=len/2;i++,j--)
        {       if(tolower(*(ptr+i))!=tolower(*(ptr+j)))
                        return 0;
        }
        return 1;
}
int main()
{  char str[25];
    int flag;
   printf("\nEnter a string:");
    scanf("%s",&str);
   flag=Ispalindrome(str);
   if(flag ==1)
        printf("\nString %s is a palindrome\n");
   else
        printf("\n String %s is not a palindrome\n");
   return 0;
}
```

Output:
```
Enter a string: sparrow
String sparrow is not a palindrome

Enter a string: malayalam
String malayalam is a palindrome
```

In the above program a string accepted from the user is passed to the function ***Ispalindrome,*** in the function it compares the first character with the last character using the pointer ***str+i*** and ***str+j*** by incrementing the value of ***i*** and decrementing the value of ***j*** and so on until either the comparison fails or succeeds. When the comparison fails, it immediately ***return 0*** indicating that it can't be a palindrome and it need not compare the rest of characters. Otherwise when it exits the loop it ***returns 1***, indicating that all the characters before and after the middle character (i.e. ***len/2***) are equal and hence it's a palindrome.

POINTERS TO POINTERS

• A variable which contains address of a pointer-variable is called pointer to a pointer.



• A variable that is a pointer to a pointer must be declared by placing an additional asterisk in front of its name.

• For example, following is the declaration to declare a pointer to a pointer of type **int**:

```
int **var;
```

• When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

```
#include<stdio.h>
void main()
{
        int var;
        int *ptr;
        int **pptr;
        var = 100; // a value of 100 is assigned to var
        ptr = &var; // stores the address of var into ptr
        pptr = &ptr; // stores the address of ptr into pptr
        printf("Value of var = %d \n", var );
        printf("Value available at *ptr = %d \n", *ptr );
        printf("Value available at **pptr = %d ", **pptr);
        return 0;
}
```

Output:

```
Value of var = 100
Value at *ptr = 100
Value at **pptr = 100
```

Advantages of pointers :

- Pointers provide direct access to memory.
- Pointers allow returning more than one value from the functions.
- Reduce the storage space and complexity of the program.
- Provides an alternate way to access array elements.
- Pointers allow us to allocate memory dynamically and also to free the memory.
- Pointers help us to build complex data structures such as linked list, stack, queues, trees, graphs etc.

Disadvantages of pointers :

- Uninitialized pointers might cause program to crash.
- Dynamically allocated memory needs to be freed explicitly. Otherwise, the memory would not be available for other programs.
- If pointers are modified with incorrect values, it may lead to memory corruption.

MEMORY ALLOCATION FUNCTIONS

• There are two types of memory allocations:

1) **Static Memory Allocation**:

• If memory-space to be allocated for various variables is decided during compilation-time itself, then the memory-space cannot be expanded to fit more data or cannot be reduced to accommodate less data.

• In this technique, once the size of the memory-space to be allocated is fixed, it cannot be altered during execution-time. This is called static memory allocation.

For ex. int a[5];

2) **Dynamic Memory Allocation**

- Dynamic memory allocation is the process of allocating memory-space during execution-time i.e. at run time.
- If storage requirement can't determined before, then the dynamic allocation technique is used.
- This allocation technique uses predefined functions to allocate and release memory for data during execution-time.

There are four library functions for dynamic memory allocation:

1) malloc()
2) calloc()
3) realloc()
4) free()

• These library functions are defined under "**stdlib.h**"

malloc()

• The name malloc stands for "memory allocation".

• This function is used to allocate a single block of memory-space during execution-time.

• The syntax is shown below:

```
data_type *p;

p=(data_type *)malloc(size);
```

here **p** is pointer variable

data_type may be int, float or char

size is number of bytes to be allocated

• If memory is successfully allocated, then address of the first byte of allocated space is returned.

If memory allocation fails, then **NULL** is returned.

• For ex:

ptr=(int*)malloc(200*sizeof(int));

• The above statement will allocate 400 bytes assuming sizeof(int)=2 bytes.

• Example: Program to find sum of N elements entered by user. Allocate memory dynamically using malloc() function.

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
        int n, i, *ptr, sum=0;
        printf("Enter number of elements: ");
        scanf("%d",&n);

        ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc

        printf("Enter elements of the array: ");

        for(i=0;i<n;++i)
        {
                scanf("%d",ptr+i);
                sum+=*(ptr+i);
        }
        printf("Sum=%d",sum);
        free(ptr);
}
```

Output:

```
Enter number of elements: 3
Enter elements of the array: 2 5 1
Sum= 8
```

calloc()

• The name calloc stands for "contiguous allocation".
• This function is used to allocate the required memory-size during execution-time and at the same time, automatically initialize memory with 0's.

• The syntax is shown below:

```
data_type *p;
p=(data_type*)calloc(int n,int size);
```

• If memory is successfully allocated, then address of the first byte of allocated space is returned. If memory allocation fails, then NULL is returned.
• The allocated memory is initialized automatically to 0's in case of integers or '\0' incase of characters.

• For ex:

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

• The above statement allocates contiguous space in memory for an array of 25 elements each of size of int, i.e. 2 bytes.
• Example: Program to find sum of **n** elements entered by user. Allocate memory dynamically using **calloc()** function.

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
        int n,i,*ptr,sum=0;
        printf("Enter number of elements: ");
        scanf("%d",&n);
        ptr=(int *)calloc(n,sizeof(int));
        printf("Enter elements of array: ");
        for(i=0;i<n;++i)
        {
                scanf("%d ",ptr+i);
                sum+=*(ptr+i);
        }
        printf("Sum=%d",sum);
        free(ptr);
}
```

Output:
Enter number of elements: 3
Enter elements of array: 2 5 1
Sum= 8

realloc()

• If the previously allocated memory is insufficient or more than sufficient. Then, previously allocated memory-size can be changed using realloc() function.

• The syntax is shown below:

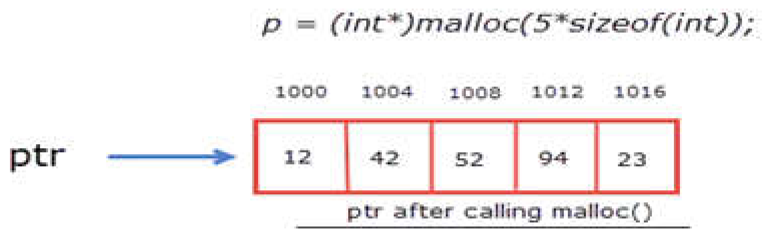
ptr=(data_type*)realloc(ptr,newsize);

• Example: Program to illustrate working of **realloc()**

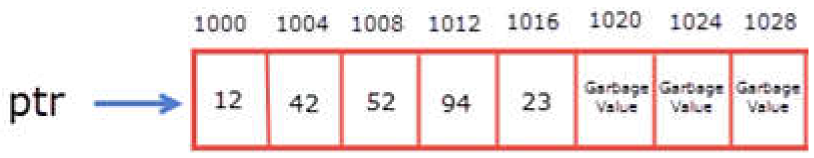
```
#include <stdio.h>
#include <stdlib.h>
void main()
{       int *ptr, i, n1, n2;
        printf("Enter size of array: ");
        scanf("%d",&n1);
        ptr=(int*)malloc(n1*sizeof(int));
        printf("Address of previously allocated memory: ");
        for(i=0;i<n1;i++)
                printf("%u\n",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

Output:

```
Enter size of array: 5
Address of previously allocated memory:
1000
1004
1008
1012
1016
```



```
Enter new size of array: 8
Address of newly allocated memory:
1000
1004
1008
1012
1016
1020
1024
1028
```



free()

• Dynamically allocated memory with either calloc() or malloc() does not release the memory allocated automatically. The programmer must use free() explicitly to release the memory space.

• The syntax is shown below:

```
free(ptr);
```

• This statement causes the space in memory pointed by **ptr** to be deallocated (or points to NULL).

Dangling pointer in C

1. A pointer is a dangling pointer whenever an object is deleted or de-allocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the de-allocated memory. In other words pointer pointing to the non-existing memory location is called the **dangling pointer**.

Examples of dangling pointer

Followingis an example of a dangling pointer.

Ex. 1

```
#include <stdio.h>
#include <stdlib.h>
void main()
{       char *ptr;
        int n;
        printf("Enter size of array: ");
        scanf("%d",&n);

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

        ……….
        ……….

        free(ptr);  //pointer now is a dangling pointer after deallocation
}
```

The pointer in the above program is a character pointer pointing a memory block, as soon as memory is de-allocated the pointer becomes a dangling pointer.

How to ensure that pointer is not a dangling pointer ?

Ex. 2

```
#include <stdio.h>
#include <stdlib.h>
void main()
{       char *ptr;
        int n;

        printf("Enter size of array: ");
        scanf("%d",&n);

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

        ……….
        ……….

        free(ptr);  //pointer now is a dangling pointer after deallocation
        ptr=NULL;
}
```

After the memory pointed to by **ptr** is freed it should be made to point to NULL i.e. it is not pointing to any memory location and hence is no more a dangling pointer.

Ex. 3

```
…
int *fun(void)
{
        int num=15;

        /* ………………*/
        return(&num);
}
```

In the above function the value and the address of num will be correct ie **some address** that was allocated to **num** and the **value as 15**. But when the function returns to the calling program the variable **num** goes out of scope and stack may have overwritten with some other value and the pointer would no longer work. However, if a pointer to **num** must be returned, num must have a scope beyond the function i.e. it may be declared as static.

Wild Pointers

Uninitialized pointers are known as wild pointers because they point to some random memory location and cause program to crash.

Ex.1

```
#include<stdio.h>
void main()
{
        int *ptr;

        *ptr=77;  // This should be avoided

}
```

In the above program pointer ptr is wild pointer because it points to no memory location and when its assigned a value of 77 it will lead to segmentation fault (core dump) error and hence which must be avoided.

Ex.2

```
#include<stdio.h>
void main()
{       int num=55;
        int *ptr;  //wild pointer
        ptr=&num; // No more a wild pointer
        *ptr=77;   // Is now fine
        .....
}
```

In the above example the pointer **ptr** is wild pointer until it is made to point to num, however after the statement ptr=&num it is no more a wild pointer and also changing the value through the pointer *ptr=77 is absolutely fine.

void pointer

The void pointer is a pointer which is not associated with any data types. It can hold address of any type and can be type casted to any type. It is also known as generic or universal pointer.

```
int x = 10;

float y = 3.14;

void *p;

p = &x;     //void pointer hold the address of x

//The following is WRONG
printf("Integer variable is = %d", *p ); //void pointer cannot be dereferenced

printf("Integer variable is = %d", *( (int*) p) );

p = &y;     //void pointer hold the address of y

printf("\nFloat variable is = %f", *( (float*) p) );
```

Also, note that **void pointer cannot be de-referenced** however it can be **explicitly type casted** to the desired type.

Advantage of void pointer

1. **malloc()** and **calloc()** returns **void *** type, which allows to allocate memory of any data type (because of void *) and hence it can be explicitly type casted to the desired type.

```
int main(void)
{
      ……………………………
      // malloc/calloc/realloc returns void * which can be
    // typecasted to int *, char *, .. or any other desired type

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

$$\begin{tikzcd}
(int\ *) \arrow[r] & \text{Explicitly typecasted to } \textbf{int *}
\end{tikzcd}$$

```
     …………………………..
}
```

STRUCTURE

INTRODUCTION

Basic data types can be used to store a single value of an entity such as person, place or thing. For ex. a person's name may be stored in a single variable of char array. But if more than one attributes needs to be stored then only a single variable can'nt solve the problem. Alternatively, a group of attributes (such as name, age, profession etc.) of different data types can be group under a common name using the concept of a structure.

What is a structure?

Structure is a ***user defined data type*** with a collection of items of different or common types grouped under a single name. Ex. Student has attributes such as Rollno, YearofAdmission and Branch etc., this all information describes a single student. While all these attributes can be grouped under the common name. The following structure would contain the attributes as follows:

Student (RollNo , Name , YearofAdmission, Branch);

The general syntax of a structure:

```
struct structure_name
{
        data_type member1;
        data_type member2;
```

```
                    data_type member3;
                };
```

In the above example **struct** is a keyword while the name of structure can be any **valid identifier**. The members of structure can be any of basic or derived data types. It can be represented in C as:

```
    struct student{
            char USN [10];
            char name[25];
            int yearofAdmission;
            char Branch[25];
            };
```

The variables declared inside the structure store values for a single entity **student** (of Engineering) and are called members of the structure. Similarly, other examples of structure include book, person, employee, company etc. The above declaration can be used to declare variables of same type as student, and hence structure is also called as user defined data type.

Structure Variable Declaration

- A structure can be declared in two ways:

1) **Tagged Structure**

- When a structure is defined, it creates a **user-defined type** and no memory space is allocated by the compiler. It acts like a template to create other variable of **student** type. For the above structure of student, members can be declared as:

```
struct student{
            char EnrollNo[10];
            char name[25];
            int yearofAdmission;
            char Branch[25];
        };
```

Inside the main function:

struct student s1, s2; // s1, s2 are structure variables of type **student**

• Here, the above statement declares the variables s1 and s2 of type **struct student**.

• Once the structure variable are declared, the compiler allocates memory for the structure
variables.

• The size of the memory allocated is not the sum of sizes of individual members. For ex. in the above example: EnrollNo: 10 x 1 byte for char =10 bytes

Name: 25 x 1 byte for char= 25 bytes
Yearof Admission: 4 bytes (assuming **int** to be of 4 bytes)= 4 bytes
Branch: 25x 1 bytes for char = 25 bytes

Therefore, total size in bytes allocated before rounding = 10+25+4+25=64 bytes. The **sizeof** for a **struct** is **not always equal to the sum of sizeof of each individual members**, because of the **padding added by the compiler to avoid alignment problem**. Padding is only added ***when a structure member is followed by a member with a larger size*** or at the end of the structure.

Therefore in the above case the largest size is of **int** ,which is 4 bytes while remaining types have 1 bytes, hence after padding the size amounts to 68 bytes.

C – Structure Padding

- In order to align the data in memory, one or more empty bytes (addresses) are inserted (or left empty) between memory addresses which are allocated for other structure members while memory allocation. This concept is called structure padding.
- Architecture of a computer processor is such a way that it can read 1 word (4 byte in 32 bit processor) from memory at a time.
- To make use of this advantage of processor, data are always aligned as 4 bytes package which leads to insert empty addresses between other member's address.
- Because of this structure padding concept in C, size of the structure is always not same as what we think.

Ex. 1 Following program demonstrates the concept of C structure padding

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

struct student
{
    int sub1;
    int sub2;
    char grade1;
    char grade2;
    float marks;
};
```

```
int main()
{
   int i;
   struct student std1 = {90, 65, 'A', 'B', 77.5};

   printf("Size of structure in bytes : %d\n", sizeof(struct student std1));

   printf("\nAddress of sub1     = %u", & std1. sub1);
   printf("\nAddress of sub2     = %u", & std1. Sub2);
   printf("\nAddress of grade1  = %u", & std1. grade1);
   printf("\nAddress of grade2  = %u", & std1. grade2);
   printf("\nAddress of marks = %u",& std1. marks);

   return 0;
}
```

OUTPUT:

```
Size of structure in bytes : 16

Address of sub1= 7567932

Address of sub2= 7567935

Address of grade1=    7567939

Address of grade2=    7567940

Address of marks = 7567943
```

Data Type	Memory allocation in C (32-bit compiler)		
	From Address	To Address	Total Bytes
int sub1	7567932	7567934	4
int sub2	7567935	7567938	4
char **grade1**	7567939		1
char **grade2**	7567940		1
Addresses 7567941 and 7567942 are padded as empty blocks			2
float marks	7567943	7567946	4

- There are 5 members declared for structure in above program Ex. 1. In 32 bit compiler, 4 bytes of memory is occupied by int datatype. 1 byte of memory is occupied by char datatype and 4 bytes of memory is occupied by float datatype.
- The table above represents the structure with memory allocation.

HOW TO AVOID STRUCTURE PADDING IN C LANGUAGE ?

#pragma pack (1) directive can be used for arranging memory of structure members which are of variable size and are very next to the end of other structure members.
The following program is same as the above program of example **Ex. 1** but demonstrates to avoid structure padding using the pre-processor directive #pragma pack(1).

Ex. 2

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

#pragma pack(1)            // avoids padding between the memory allocated to the
                           //members

struct structure1
{
     int id1;
     int id2;
     char name;
     char c;
     float percentage;
};

int main()
{
   struct structure1 a;

   printf("Size of structure1 in bytes : %d\n",  sizeof(a));
   printf ( "\n   Address of id1        = %u", &a.id1);
   printf ( "\n   Address of id2        = %u", &a.id2);
   printf ( "\n   Address of name       = %u", &a.name);
   printf ( "\n   Address of c        = %u", &a.c);
   printf ( "\n   Address of percentage = %u", &a.percentage );
   return 0;
}
```

OUTPUT:

```
 Size of structure in bytes : 14
Address of id1       = 4143303844
Address of id2       = 4143303848
Address of name      = 4143303852
Address of c         = 4143303853
Address of percentage = 4143303854
```

2) **Type Defined Structure**

• Another way of creating structure variable using the keyword **typedef** is:

```
typedef struct student
{
            char USN [10];
            char name[25];
            int yearofAdmission;
            char Branch[25];
}STD;

OR

typedef struct {
            char USN [10];
            char name[25];
            int yearofAdmission;
            char Branch[25];
}STD1,STD2;
```

Inside the **main** function:

```
STD s1,s2 ;
```

• Here the above statement declares that s1 and s2 are variables of type STD (which is of type struct student).

Structure Variable Initialization

- Using the tagged method. For ex:

```
struct student{
            char USN [15];
            char name[25];
            int yearofAdmission;
            char Branch[25];
        };
```

```
struct student s1={“PV-1012050”,"Suresh",2020, “CSE”};

OR
struct {
        char USN [15];
        char name[25];
        int yearofAdmission;
        char Branch[25];
    }student={“PV-1012043”,"Somesh",2020, “EC”};
```

- Using the typedef method. For ex:

```
typedef struct student {
        char USN [15];
        char name[25];
        int yearofAdmission;
        char Branch[25];
    }STD;

OR

typedef struct{
        char USN [15];
        char name[25];
        int yearofAdmission;
        char Branch[25];
    }STD;

STD s1= {“ PV-1012065”,"Ramesh",2020, “ME”};
```

Accessing Members of a Structure

The structure members can be accessed by using the member operator(.) dot operator i.e structure variable name followed by dot operator followed by the name of the structure member.

To access members of the student using the structure variable s1 in the above example:

- **s1.USN:** accesses USN of any single student i.e. 2MM16CS001
- **s1.name:** accesses Name of any single student i.e. Suresh
- **s1.yearofAdmission**: accesses admission year of any single student i.e. 2016
- **s1.Branch:** accesses the branch of engineering of any single student i.e. CSE
- The values can be read into the members of the structure as:

```
gets(s1.name); or scanf("%s", s1.name);
scanf("%d",&s1.yearofAdmission);
```

- The above values can be printed as:

```
printf("%s", s1.name);
printf("%d", s1.yearofAdmission);
```

ARRAY OF STRUCTURES

The above definition of structure allows only a single instance of a variable of type structure. But, practically the problem demands to store & process the information of more than one student. In that case the structure variable can be declared with a size i.e by declaring it as an array of structure.

For ex. **struct student s1[50];** //declares s1 as an array of structure that can store
//information of 50 students.

OR

STD s1[50]; //declares s1 same as above.

To access the structure member using the array of structure:

```
for(i=0;i<5; i++)
{       printf("\n Enter the USN:");
        scanf("%s",&s1[i].USN);
        printf("\n Enter the Name:");
        scanf("%s",&s1[i].name);
        printf("\n Enter the Year of Admission:");
        scanf("%s",&s1[i].yearofAdmission);
        printf("\n Enter the branch name:");
        scanf("%s",&s1[i].branch);
}
```

The above code will accept information of five students & store it in the array of structure variable s1.

To print the information of five students accepted above.

```
printf("\n The students details are:\n");
for(i=0;i<5; i++)
{       printf("\nUSN:%s",s1[i].USN);
        printf("\nName:%s",s1[i].name);
        printf("\nYear of Admission:%s",s1[i].yearofAdmission);
        printf("\n Branch name:%s\n\n",s1[i].branch);
        }
```

STRUCTURES WITHIN STRUCTURES

Whenever a structure is defined within another structure it's called as nested structure. Suppose the date of admission of a student is stored in another structure the earlier structure could be rewritten as:

• For ex:

```
struct DOA   //DOA: Date of Admission
{
        int day;
        int month;
        int year;
};

struct student
{
        char USN [10];
        char name[25];
        struct DOA adm;
        char Branch[25];
};
```

The structure declaration in the main function:

```
        struct student s1 ,s2;
```

• To access the date of admission year for any student, then it would be accessed as:

```
        s1.adm.year
```

Structure with a pointer as the member of the structure

Consider the following program declared with a structure containing Publisher as the pointer member.

```
struct Book
{
        char BName[20];
        char BAuthor[25];
        char *Publisher;
        float price;
};
void main()
{
        struct Book  bok={“Programming in ANSI C”,”E Balagurusamy”,”TMH”,360};

        printf("\nBook Name: %s\n",bok.BName);
        printf("Author Name: %s\n",bok.BAuthor);
        printf("Publisher Name: %s\n",bok.Publisher);
        printf("Price:Rs. %0.2f",bok.price);
}
```

OUTPUT:

```
Book Name: Programming in ANSI C
Author Name: E Balagurusamy
Publisher Name: TMH
Price: Rs. 360.00
```

STRUCTURES AND FUNCTIONS

- structure can be passed to functions in following ways:
    - Passing by value (passing actual value as argument)
    - Passing by reference (passing address as an argument)

Passing Structure by Value

- A structure-variable can be passed to the function as a normal variable.
- If structure is passed by value, changes made to the structure-variable inside that function does not reflect those changes in the calling-function.
- Example: To demonstrate passing a structure by value.

```
#include<stdio.h>
struct employee
{
        char name[50];
        float salary;
};
void DisplayInfo(struct employee emp)  //passes the structure emp by value
{
        printf("Employee Name: %s \n",emp.name);
        printf("\nHis Salary: %d",std.salary);
}
void main()
{
        struct student emp;
        printf("Enter Employee’s Name: ");
        scanf("%s", &emp.name);
        printf("Enter Salary: ");
        scanf("%f", &emp.salary);
        DisplayInfo(emp);
}
```

```
Output:
Enter Employee’s Name: Ramesh
Enter Salary:  75000
Employee Name: Ramesh
His Salary:  75000
```

Passing Structure by Reference

- The address of structure variable is passed to the function.
- If structure is passed by reference, changes made to the structure variable in the function call are also reflected in the calling-function.
- Example: Following program to multiply two integers entered by user and then store their product into another variable.

The following program demonstrates the above concept, a structure containing four members x, y, remainder and quotient is declared. Then, the structure variable **calc** of **struct Calculate** type is passed to the function **by reference,** then the remainder and quotient is computed by accessing the x and y members of the structure and stored it in other members quotient and remainder respectively. Then finally they are displayed in main program.

```
#include <stdio.h>
struct Calculate
{       int x,y;
        int remainder, quotient;
};

void Compute(struct product *p)
{
   p->quotient=(p->x) /(p->y) ;            //Finds the Quotient by dividing of x by y
   p->remainder=(p->x) %(p->y) ;        // Finds the remainder by performing x mod y
 }
void main()
{
        struct Calculate calc;
        printf("\nEnter the first number:");
        scanf("%d",&calc.x);
        printf("\nEnter the second number: ");
        scanf("%d",&calc.y);
        Compute(&calc);                 //passes opr structure variable by reference
        printf("\nQuotient is %d", calc.quotient);
        printf("\nRemainder is %d", calc.remainder);
}
```

OUTPUT:
Enter the first number: 100
Enter the second number: 3
Quotient is 33
Remainder is 1

Self referential structure

Consider the following example where a structure member a pointer refers to itelf.

Ex. 1:

```
#include < stdio.h >

struct node
{       int data1;
        char data2;
        struct node *link;    //pointer to the same structure type;
};
void main()
{
        struct node objNode={100,’d’,NULL};
        if(objNode.link ==NULL)
        {
                objNode.link=&objNode;                  //points to the objNode
                printf(“%d”,objNode.link->data1);       //data members are accessed with an
                                                        //arrow thru the pointers
        }
}
```

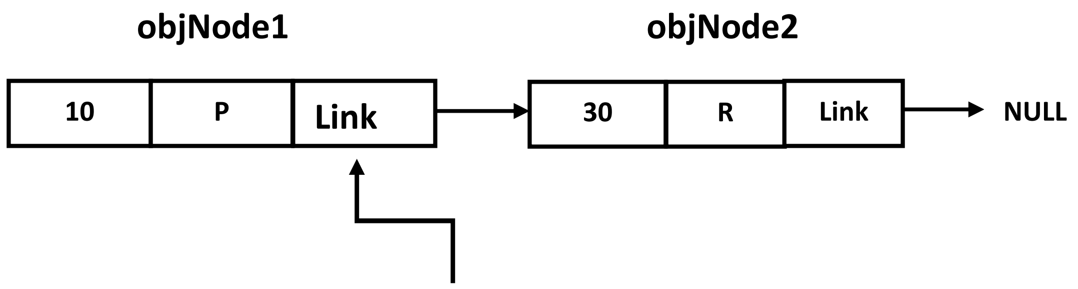
OUTPUT:
100

Ex. 2:

```
struct node {
        int data1;
        char data2;
        struct node* link;
};

int main()
{
        struct node objNode1;  // Node1

        // Initialization
        objNode1.link = NULL;
        objNode1.data1 = 10;
        objNode1.data2 = 'P';

        struct node objNode2; // Node2

        // Initialization
        objNode2.link = NULL;
        objNode2.data1 = 30;
        objNode2.data2 = 'R';

        // Linking ob1 and ob2
        objNode1.link = &ob2;

        // Accessing data members of ob2 using ob1
        printf("%d", objNode1.link->data1);
        printf("\n%d", objNode1.link->data2);
        return 0;
}
```

OUTPUT:

```
30
R
```



Points to the address of objNode2 structure

Union

Union is a user defined data type and unlike structures all the members share the same memory location. Compiler allocates storage based on the largest size of the member.

Ex. 1

```
union Device
{
        int port;
        int code;
};
```

```
int main()
{       union Device dvr;
        dvr.port=65;
        printf("port= %d code= %d\n",dvr.port,dvr.code);
        printf("Size of Device = %d bytes",sizeof(Device));
        return 0;
}
```

OUTPUT:

```
Port= 65 code= 65
Size of Device = 4 bytes
```

In the example Ex.1 declaration of union Device, all the members' port and code share the same memory location of 4 bytes (on 32-bit machine). Any changes done to any member will have the same effect on other members as well. For ex. if value of port is set to 65 then code will also have the same value 65.

Difference between union and structure

a)

```
struct Device
{       int port;
        char code;
};
```

b)

```
union Device
{       int port;
        char code;
};
```

For the declaration in (a) struct Device, compiler reserves 8 bytes while in (b) it allocates 4 bytes the largest size of the member **port** is of integer type.

Below figure shows Memory allocation of 8 bytes for **struct** in Fig. (a) and Fig. (b) shows 4 bytes of memory allocated to union that is the largest member port of **int** type.

int (4)	**char (4)**

Fig. (a) struct

int/char (4)

Fig. (b) union

Pointers to unions

Pointers to unions can be declared same as pointers to structures, while the union members can be accessed using the arrow operator (→).

```
union Device
{
        int port;
        char code;
};

int main()
{       union Device dvr;
        union Device *ptr;
        dvr.port=71;
        ptr=&dvr;
        printf("\nThe value through pointer:%d %c",(ptr->port), (ptr->code));
        return 0;
}
```

OUTPUT:

```
The value through pointer: 71 G
```