UNIT-2

- Pointers –Basic of pointers and addresses, Pointers and arrays, Pointer arithmetic, passing pointers to functions, call by reference,
- Dynamic memory management in C - malloc(), calloc(), realloc(), free(), memory leak,
- Dangling, Void, Null and Wild pointers
- Structures - Structures, array of structures, structure within structure, union, typedef, self-referential structure, pointer to structure

UNDERSTANDING THE COMPUTER'S MEMORY

- Our data and programs need to be placed in the primary memory for execution.
- The primary memory or RAM (Random Access Memory) is a collection of memory locations (often known as cells) and each location has a specific address. Each memory location is capable of storing 1 byte of data
- Generally, the computer has three areas of memory each of which is used for a specific task. These areas of memory include- stack, heap and global memory.

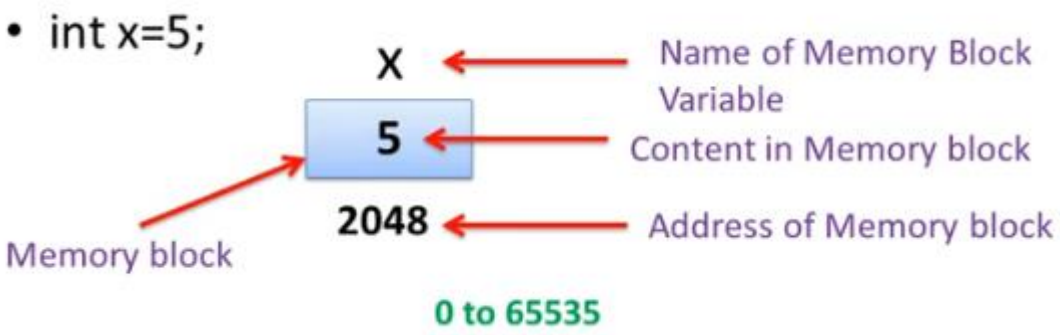
- Pointer

Pointer is a variable that contains address of another variable.

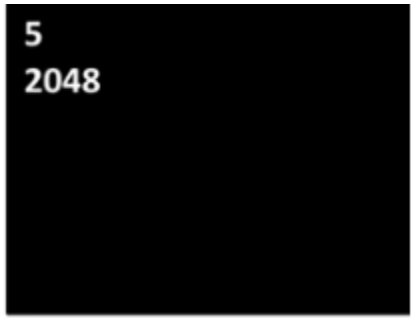
Pointer always consumes 2 bytes in memory.

Lets take an example:

```
#include<stdio.h>
void main()
{
int x=5;
printf("%d",x);
printf("\n %u",&x);
}
```



Output:



- %d -32768 to 32767
- %u 0 to 65535

Address of operator:

- & is known a address of operator
- It is an unary operator, and its operand must be the name of variable.
- & is also known as referencing operator.

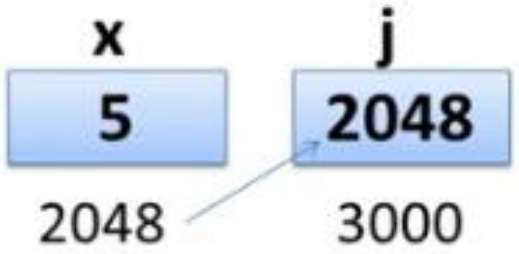
value at address' operator

- The other pointer operator available in C is **'*', called 'value at** address' operator. It gives the value stored at a particular address.
- The 'value at address' operator is also called 'indirection' operator.
- Note that printing the value of *(&x) is same as printing the value of x.

Pointers in C

Pointer

```
int *j, x=5;
j=&x;
```

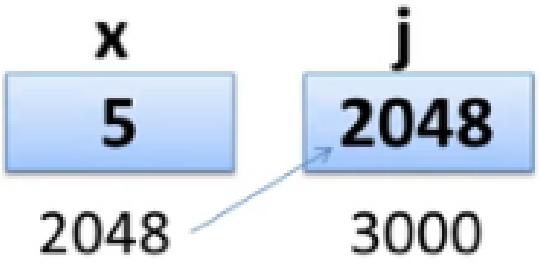


- j is not an ordinary variable like any other integer variable.
- It is a variable which contains the address of another variable

- Pointers in C

Question

```
main()
{
 int x=5, *j;
 j=&x;
 printf("%d %u\n", x, j);
 printf("%d %u ", *j, &x);
 printf(" %u", *&j);
}
```



OutPut

```
5 2048
5 2048
2048
```

Declaring Pointer Variable

- Actually pointers are nothing but memory addresses.
- A pointer is a variable that contains the memory location of another variable.
- The general syntax of declaring pointer variable is

data_type *pointer_name;

For example:

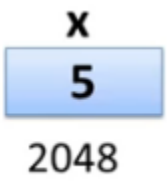
```
int *p
int x= 10;
int *p = &x;
```

The ***** informs the compiler that **p** is a pointer variable and the **int** specifies that it will store the address of an integer variable.

The **&** operator retrieves the lvalue (address) of **x**, and copies that to the contents of the pointer **p**.

Question

```
int x=5;
&x=7;
```



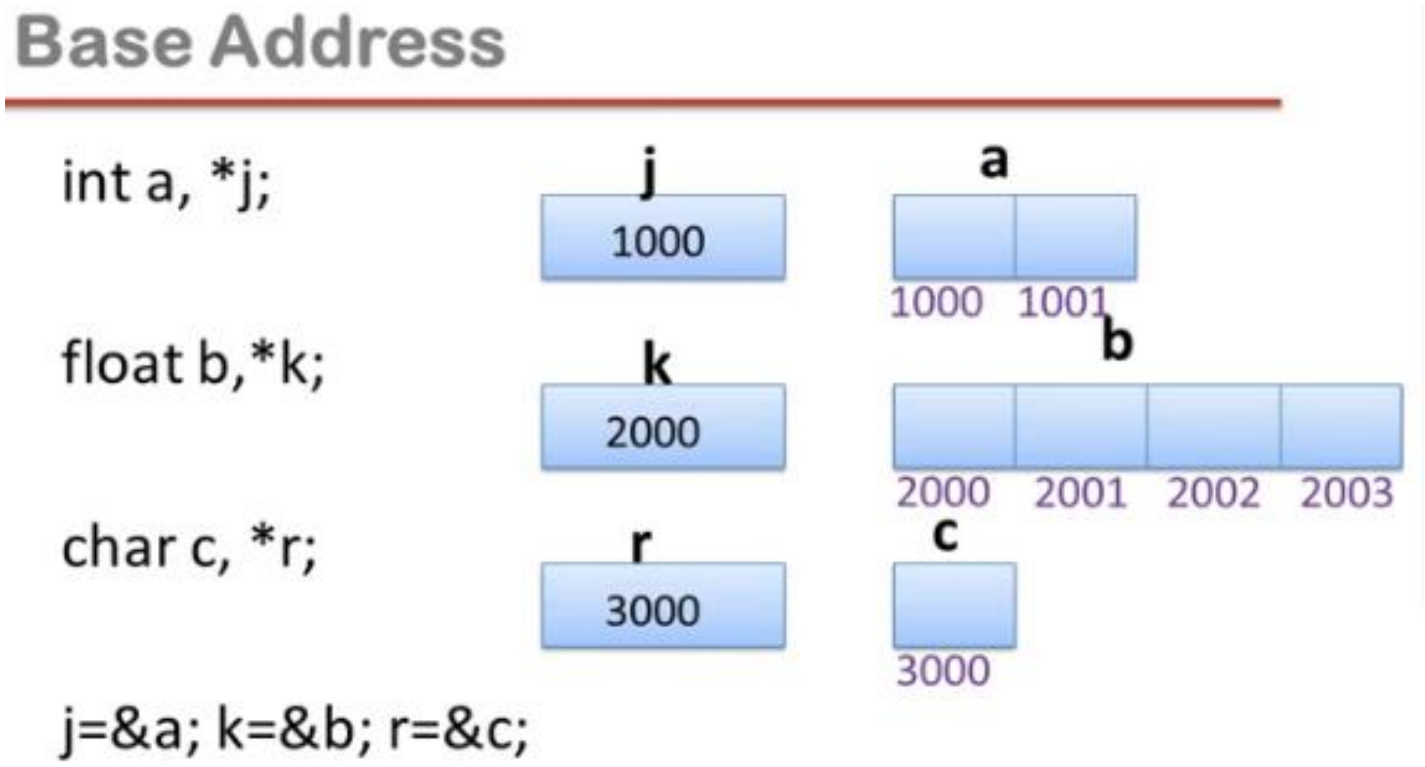
We cannot store anything in &x as &x is not a variable, it is the way to represent address of block x

```
int x=5;
int *j;
j=&x;
```

We can store address in another variable

But j has to be declared before use.

Pointers in C



De-referencing a Pointer Variable

- We can "dereference" a pointer, i.e. refer to the value of the variable to which it points by using unary '*' operator as in *ptr.

  That is, *ptr = 10, since 10 is value of x.

```
 #include<stdio.h>
int main()
{
        int num, *pnum;
        pnum = &num;
        printf("\n Enter the number :" );
        scanf("%d", &num);
        printf("\n The number that was entered is : %d", *pnum);
        return 0;
}
```

OUTPUT:

Enter the number : 10

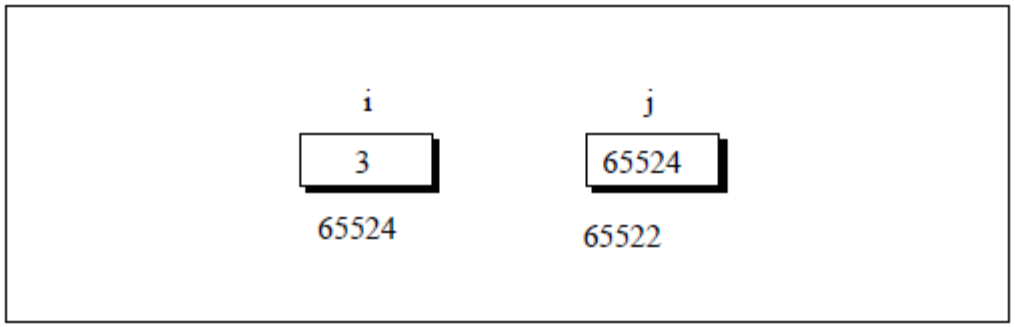
The number that was entered is : 10

Lets take an Example:

```
int i=3; *j;
j = &i ;
```

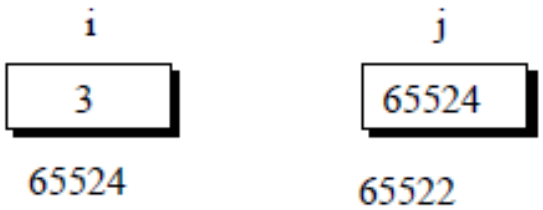
Where i's value is 3 and j's value is i's address.

The expression **&i gives the address of the variable i. This address** can be collected in a variable.



- But remember that **j is not an ordinary variable like any other** integer variable. It is a variable that contains the address of other variable**.**

  Since j is a variable the compiler must provide it space in the memory.




```
main( )
{
int i = 3 ;
int *j ;
j = &i ;
printf ( "\nAddress of i = %u", &i ) ;
printf ( "\nAddress of i = %u", j ) ;
printf ( "\nAddress of j = %u", &j ) ;
printf ( "\nValue of j = %u", j ) ;
printf ( "\nValue of i = %d", i ) ;
printf ( "\nValue of i = %d", *( &i ) ) ;
printf ( "\nValue of i = %d", *j ) ;
}
```

OUTPUT:

Address of i = 65524
Address of i = 65524
Address of j = 65522
Value of j = 65524
Value of i = 3
Value of i = 3
Value of i = 3

- **Pointer to Pointer**

  We can store the address of a pointer variable in some other variable, which is known as a pointer to pointer variable.

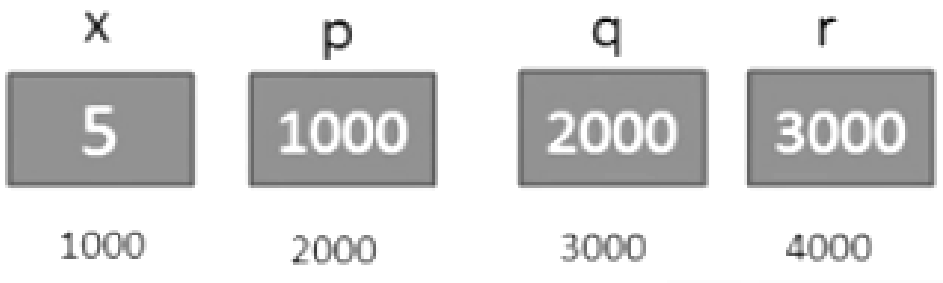
For ex.

```
int **ptr;
```

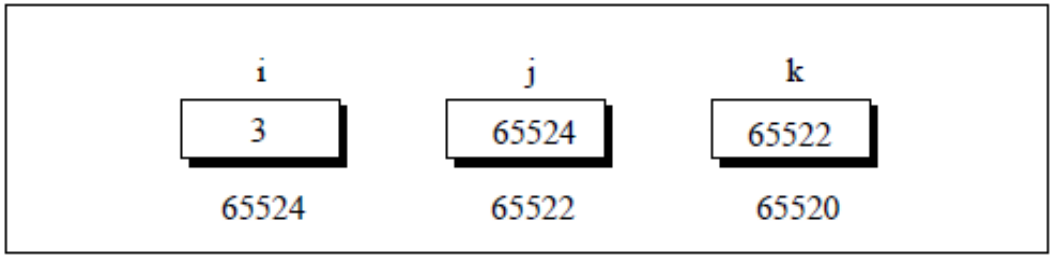
- Here variable ptr is a pointer to pointer and it can point to a pointer pointing to a variable of type int.
- The double asterisk used in the declaration informing the compiler that a pointer to pointer is being declared.

For Example

```
void main( )
{
    int x=5,*p,**q,***r;
    p=&x;
    q=&p;
    r=&q;
}
```



For Example:




```
main( )
{
int i = 3, *j, **k ;
j = &i ;
k = &j ;
printf ( "\nAddress of i = %u", &i ) ;
printf ( "\nAddress of i = %u", j ) ;
printf ( "\nAddress of i = %u", *k ) ;
printf ( "\nAddress of j = %u", &j ) ;
printf ( "\nAddress of j = %u", k ) ;
printf ( "\nAddress of k = %u", &k ) ;
printf ( "\nValue of j = %u", j ) ;
printf ( "\nValue of k = %u", k ) ;
printf ( "\nValue of i = %d", i ) ;
printf ( "\nValue of i = %d", * ( &i ) ) ;
printf ( "\nValue of i = %d", *j ) ;
printf ( "\nValue of i = %d", **k ) ;
}
```

The output of the above program would be:

```
Address of i = 65524
Address of i = 65524
Address of i = 65524
Address of j = 65522
Address of j = 65522
Address of k = 65520
Value of j = 65524
Value of k = 65522
Value of i = 3
Value of i = 3
Value of i = 3
Value of i = 3
```

Pointer to an Array

//Write a program to read and display an array of n integers

```
#include<stdio.h>
int main()
{
    int  arr[5] = {1, 2, 3, 4, 5};
    int *ptr = &arr[0];
ptr++;
printf("\n The value of the second element of the array is %d", *ptr);
}
```

OUTPUT:

The value of the second element of the array is 2

```
//Write a program to read and display an array of n integers
#include<stdio.h>
int main()
{
    int  i, arr[5] = {1, 2, 3, 4, 5};
    int *ptr;
    for(i=0;i<5;i++)
    {
    ptr =&arr[i];
    printf("\n value is %d", *ptr);
    ptr++;
    }
}
```

OUTPUT:
value is 1
value is 2
value is 3
value is 4
value is 5

```
//Write a program to read and display an array of n integers
#include<stdio.h>
int main()
{
    int i, n;
    int arr[10], *p;
    p = arr;
    printf("\n Enter the number of elements : ");
    scanf("%d", &n);
    for(i=0; i <n; i++)
                    scanf("%d", (p+i));
    for(i=0; i <n; i++)
        printf("\n arr[%d] = %d", i, *(p+i));
}
```

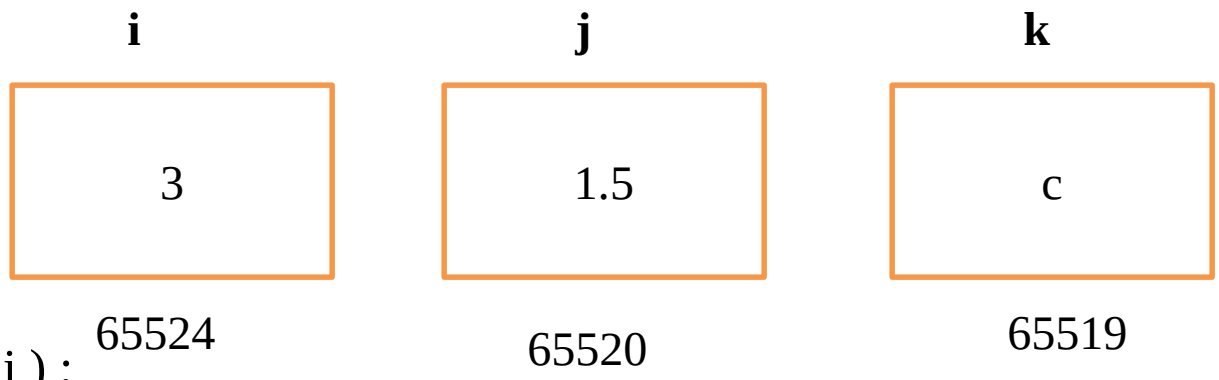
```
 Enter the number
of elements : 5
11
22
33
44
55

 arr[0] = 11
 arr[1] = 22
 arr[2] = 33
 arr[3] = 44
 arr[4] = 55
------------------------------
```

Pointer's Arithmetic:

- We cannot add, multiply or divide two addresses(Subtraction is possible)
- We cannot multiply an integer to an address and similarly we cannot divide an address with an integer value
  - We can add or subtract integer to/from an address

Example:

```
main( )
{
int i = 3, *x ;
float j = 1.5, *y ;
char k = 'c', *z ;
printf ( "\nValue of i = %d", i ) ;
printf ( "\nValue of j = %f", j ) ;
printf ( "\nValue of k = %c", k ) ;
x = &i ;
y = &j ;
z = &k ;
printf ( "\nOriginal address in x = %u", x ) ;
printf ( "\nOriginal address in y = %u", y ) ;
printf ( "\nOriginal address in z = %u", z ) ;
x++ ;
y++ ;
z++ ;
printf ( "\nNew address in x = %u", x ) ;
printf ( "\nNew address in y = %u", y ) ;
printf ( "\nNew address in z = %u", z ) ;
}
```



OUTPUT:

```
Value of i = 3
Value of j = 1.500000
Value of k = c
Original address in x = 65524
Original address in y = 65520
Original address in z = 65519
New address in x = 65526
New address in y = 65524
New address in z = 65520
```

The way a pointer can be incremented, it can be decremented as well, to point to earlier locations.

Thus, the following operations can be performed on a pointer:

(a) Addition of a number to a pointer.

For example,

```
int i = 4, *j, *k ;
j = &i ;
j = j + 1 ;
j = j + 9 ;
k = j + 3 ;
```

b) Subtraction of a number from a pointer. For example,

```
int i = 4, *j, *k ;
j = &i ;
j = j - 2 ;
j = j - 5 ;
k = j - 6 ;
```

Subtraction of one pointer from another:

One pointer variable can be subtracted from another provided both variables point to elements of the same array. This is illustrated in the following program.

```
main( )
{
int arr[ ] = { 10, 20, 30, 45, 67, 56, 74 } ;
int *i, *j ;
i = &arr[1] ;
j = &arr[5] ;
printf ( "%u %u %u %d",i, j, j-i, *j-*i ) ;
}
```

OUTPUT:

2293408 2293424 4 36

Comparison of two pointer variables:

Pointer variables can be compared provided both variables point to objects of the same data type. It can be useful when both pointer variables point to elements of the same array.

```
main( )
{
int arr[ ] = { 10, 20, 36, 72, 45, 36 } ;
int *j, *k ;
j = &arr[ 4 ];
k = ( arr + 4 );
if (j == k)
printf ( "The two pointers point to the same location" ) ;
else
printf ( "The two pointers do not point to the same location" ) ;
}
```

OUTPUT:

The two pointers point to the same location

Do not attempt the following operations on pointers... they would never work out.

(a) Addition of two pointers

(b) Multiplication of a pointer with a constant

(c) Division of a pointer with a constant

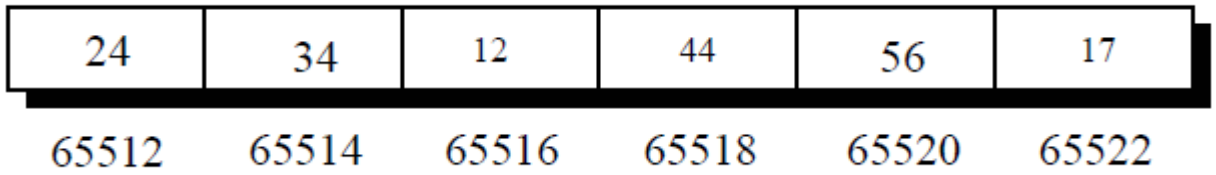
Now we will try to correlate the following two facts:

(a) Array elements are always stored in contiguous memory locations.

(b) A pointer when incremented always points to an immediately next location of its type.

- For example: (program that will print the memory locations in which the elements of this array are stored.)

num[ ] = { 24, 34, 12, 44, 56, 17 }




```
main( )
{
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int i ;
        for ( i = 0 ; i <= 5 ; i++ )
        {
        printf ( "\nelement no. %d ", i ) ;
        printf ( "address = %u", &num[i] ) ;
        }
}
```

output:

element no. 0 address = 65512
element no. 1 address = 65514
element no. 2 address = 65516
element no. 3 address = 65518
element no. 4 address = 65520
element no. 5 address = 65522

Back to Function Calls

- the two types of function calls:
  - call by value and
  - call by reference

Arguments can generally be passed to functions in one of the two ways:

(a) sending the values of the arguments

(b) sending the addresses of the arguments

In the first method the 'value' of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function.

- Example (Call by Value): The following program illustrates the 'Call by Value'.

```
main( )
{
    int a = 10, b = 20 ;
    swapv( a, b ) ;
    printf ( "\na = %d b = %d", a, b ) ;
}

swapv( int x, int y )
{
    int t ;
    t = x ;
    x = y ;
    y = t ;
    printf ( "\nx = %d y = %d", x, y ) ;
}
```

Note that values of **a and b remain unchanged even after** exchanging the values of **x and y.**

Output
x = 20 y = 10
a = 10 b = 20

- call by reference :
- In the second method (call by reference) the addresses of actual arguments in the calling function are copied into formal arguments of the called function.

  This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them.

- Example (call by reference):

```
main( )
{
    int a = 10, b = 20 ;
    swapr ( &a, &b ) ;
    printf ( "\na = %d b = %d", a, b ) ;
}
        swapr( int *x, int *y )
        {
        int t ;
        t = *x ;
        *x = *y ;
        *y = t ;
        }
```

Output:
a = 20 b = 10

Note that this program manages to exchange the values of **a and b** using their addresses stored in **x and y.**

- Usually in C programming we make a call by value. This means that in general you cannot alter the actual arguments. But if desired, it can always be achieved through a call by reference.

Conclusions

From the programs that we discussed here we can draw the following conclusions:

(a) If we want that the value of an actual argument should not get changed in the function being called, pass the actual argument by value.

(b) If we want that the value of an actual argument should get changed in the function being called, pass the actual argument by reference.

(c) If a function is to be made to return more than one value at a time then return these values indirectly by using a call by reference.

Passing an Array to a Function

```
/* Demonstration of passing an entire array to a function */
void display(int *, int);
void main( )
{
int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
dislpay ( &num[0], 6 );        // can be written as: display (num, 6 );
}

void display( int *j, int n )
{
     int i;
     for (i=0;i<n;i++)
     {
     printf("\n element= %d",*j);
     j++;                    /* increment pointer to point to next element */
     }
}
```

Example:

```
/* Accessing array elements */
main( )
{
int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
int i ;
for ( i = 0 ; i <= 5 ; i++ )
{
printf ( "\n element = %d %d ", num[i], *( num + i ) ) ;
printf ( "\t %d %d", *( i + num ), i[num] ) ;
}
}
```

OUTPUT

```
 element = 24 24        24 24
 element = 34 34        34 34
 element = 12 12        12 12
 element = 44 44        44 44
 element = 56 56        56 56
 element = 17 17        17 17
-----------------------------
```

- Null Pointers

NULL Pointer is a pointer which is pointing to nothing.

In case, if we don't have address to be assigned to a pointer, then we can simply use NULL.

To declare a null pointer you may use the predefined constant NULL,

for ex:

```
int main()
{
    int *ptr = NULL;       // Null Pointer
    printf("The value of ptr is %p", ptr);
    return 0;
}
```

Output :

```
The value of ptr is (nil)
```

Another Example of Null Pointers

```
#include <stdio.h>
int main()
{
    int *i= NULL, *j= NULL;
            if(i == j)
            {
            printf("\n both i and j are same");
            printf("\n %u %u",i,j);
            //printf("\n %d %d",*i,*j);
            }
return 0;
}
```

OUTPUT:

both i and j are same

0 0

- Generic Pointers or Void pointer
- A generic or void pointer is pointer variable that has void as its data type. The generic pointer, can be pointed at variables of any data type.
- For ex:

```
void main()
    {       int x=10;
            char ch = 'A';
            void *p;
            p = &x;
            printf("\n void pointer points to the int value=%d", *(int*)p);
            p = &ch;
            printf("\n void pointer now points to the character %c", *(char*)p);
    }
```

OUTPUT:

Generic pointer points to the integer value = 10

Generic pointer now points to the character = A

For example: (Generic Pointers or Void pointer)

```
main()
{
    int x = 4;
    float y = 5.5;
    void *p;  //A void pointer
    p = &x; // void pointer is now int
    printf("Integer variable is = %d", *( (int*) p) );

    p = &y; // void pointer is now float
    printf("\nFloat variable is= %f", *( (float*) p) );
}
```

// (int*)p does type casting of void
// *((int*)p) dereferences the typecasted

Output:

```
Integer variable is = 4
Float variable is= 5.500000
```

Wild Pointer in C

Uninitialized pointers are known as wild pointers.

A pointer which has not been initialized to anything (not even NULL) is known as wild pointer.

Certain size of memory is provided to our program, that can be further divided into Free memory area and consumed memory area.

```
int main()
{
    int *p;          /* wild pointer */
/* Some unknown memory location is being corrupted.  This should never be done. */
}
```

- Please note that if a pointer p points to a known variable then it's not a wild pointer. In the below program, p is a wild pointer till this points to a.

```
int main()
{
int *p;     /* wild pointer */
int a = 10;
p = &a;     /* p is not a wild pointer now*/
*p = 12;   /* This is fine. Value of a is changed */
}
```

- Dangling pointer

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.

There are **three** different ways where Pointer acts as dangling pointer, these are: De-allocation of memory, Function Call and Variable goes out of scope

```
void fun(void)
main()
{
fun();
}
        void fun(void)
        {
        int *p;
                {
                int x=10;
                p=&x;
                printf("%u",p);
                p=null;
                }
        printf("%u",*p);
        }
```