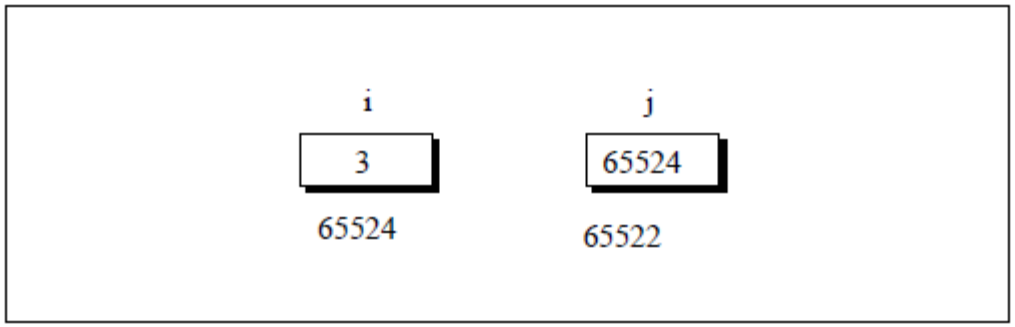
Lets take an Example:

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

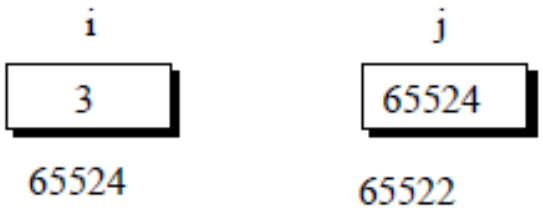
Where i's value is 3 and j's value is i's address.

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.

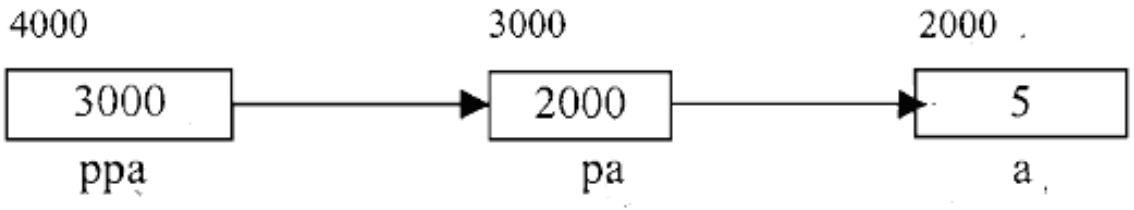
The syntax of declaring a pointer to pointer is as-

data_type **pptr;

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

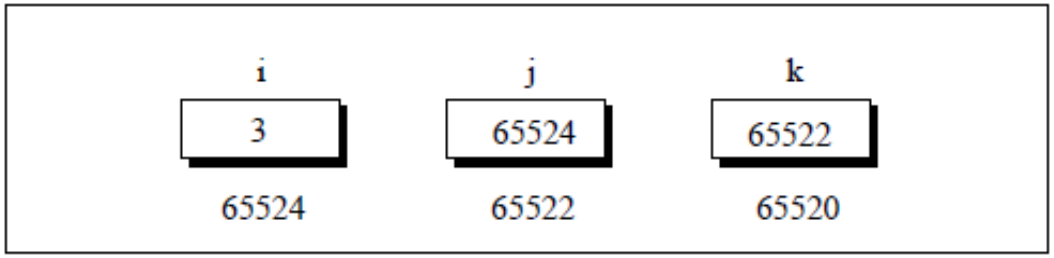
- Example

```
int a = 5;
int *pa = &a;
int **ppa = &pa;
```



Here type of variable a is int, type of variable pa is (int *) or pointer to int, and type of variable ppa is (int **) or pointer to pointer to int.

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 ) ;
}
```

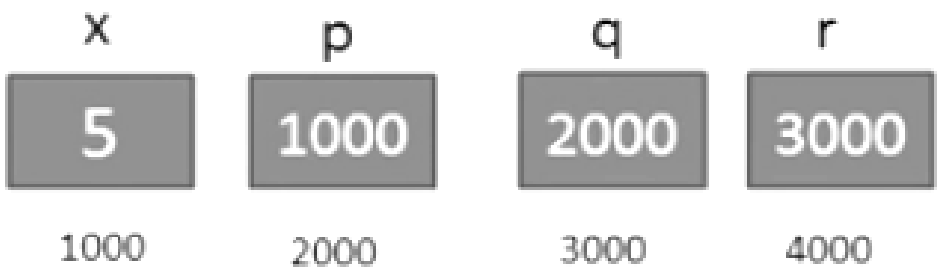
The output of the above program would be:

```
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
```

- In the above example, Here, i is an ordinary int, j is a pointer to an int (often called an integer pointer), whereas k is a pointer to an integer pointer.

For Example

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



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

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

- 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.

- NULL POINTERS

A *null pointer* which is a special pointer value that is known not to point anywhere. This means that a NULL pointer does not point to any valid memory address.

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

```
int *ptr = NULL;
```

You can always check whether a given pointer variable stores address of some variable or contains a null by writing,

```
if ( ptr == NULL)
{       Statement block;
}
```

Null pointers are used in situations if one of the pointers in the program points somewhere some of the time but not all of the time. In such situations it is always better to set it to a null pointer when it doesn't point anywhere valid, and to test to see if it's a null pointer before using it.

GENERIC POINTERS

- A generic pointer is pointer variable that has void as its data type. The generic pointer, can be pointed at variables of any data type. It is declared by writing

  void *ptr;

- You need to cast a void pointer to another kind of pointer before using it. Generic pointers are used when a pointer has to point to data of different types at different times. For ex,

```
int main()
    {       int x=10;
            char ch = ‘A’;
            void *gp;
            gp = &x;    printf("\n Generic pointer points to the integer value = %d", *(int*)gp);

            gp = &ch;  printf("\n Generic pointer now points to the character %c", *(char*)gp);
            return 0;
     }
```

OUTPUT:

Generic pointer points to the integer value = 10

Generic pointer now points to the character = A

PASSING ARGUMENTS TO FUNCTION USING POINTERS

- The calling function sends the addresses of the variables and the called function must declare those incoming arguments as pointers. In order to modify the variables sent by the caller, the called function must dereference the pointers that were passed to it. Thus, passing pointers to a function avoid the overhead of copying data from one function to another.

```
#include<stdio.h>
    int main()
    {
        int num1, num2, total;
        printf("\n Enter two numbers : ");
        scanf("%d %d", &num1, &num2);
        sum(&num1, &num2, &total);
        printf("\n Total = %d", total);
        return 0;
    }

    void sum ( int *a, int *b, int *t)
    {
        *t = *a + *b;
    }
```

//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);
}
```

```
//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));
}
```

Now, consider the following program that prints a text.

```
#include<stdio.h>
main()
{   char str[] = “Oxford”;
        char *pstr = str;
    printf(“\n The string is : “);
    while( *pstr != ‘\0’)
    {     printf(“%c’, *pstr);
        pstr++;
    }
}
```

- In this program we declare a character pointer *pstr to show the string on the screen. We then "point" the pointer pstr at str. Then we print each character of the string in the while loop. Instead of using the while loop, we could have straight away used the function puts(), like

  puts(pstr);

- Consider here that the function prototype for puts() is:

  int puts(const char *s);

Here the "const" modifier is used to assure the user that the function will not modify the contents pointed to by the source pointer. Note that the address of the string is passed to the function as an argument.