Pointers

We will continue our discussion on the pointers and how we put this concept to use.

We have already seen the representation of pointers in a program, i.e. how do we declare pointers and how to get the address to which it is pointing to. For example to declare a pointer "num" of the type "integer" we used "int *num;". we also saw that, to assign or to point this pointer to another integer variable "n" we will use the statement "num=&n;" .

Here we created a pointer called num and then directed this num to the address of the number n.Thats what the use of "&" operator is . It returns the address of the argument "n" and then assigns that to "num ".

Now we will see the use of * operator. We are going to assign the value stored in the address, to which "num" is pointing , to another variable "m" . The way that is done is through the staement, "m=* num;" . Here * returns the data to which "num " is pointing to.

Remember that "*" and "&" are two operators that is closely associated with a pointer.Then we come to arrays. The syntax to declare an integer array "m" of four elements is int x[4]; The type here is integer and array dimension is 4 . Now elements of the array would be x[0],x[1],x[2] and x[3]. Another important thing to note is that we use square brackets instead of () brackets to mark the elements of the array. This is because we use () to represent functions.

Now we will see how one can use arrays and pointers interchangeably. Let us look at the following code.

```
int *num;

int x[4];

num=x;
```

Here we have an integer pointer declared num, which is of type int and an integer array x . By saying num=x we are point num to the base address of the array x. i.e to the base element of x.

If we now write num++ it will point to the next element of the array x i.e. x[1].

There are two important things to take note of. We can point num to the " i^{th}"element of the array by stating num=x[i]; . num++; will then point to the "$(i+1)^{th}$ " element. The second point is that num++; can be used recursively to point to the successive elements of the array. i.e.,

```
num=x;

for(i=0;i<=3;i++)

 {num++;}
```

will make num point to all the elements of the array successively and we can get the value stored in the array elements by using "*num". However if we look at the address of "num" it will always be the base address of "x".

Printing the address of array elements using pointers:

Pointers can be used to get the address of each element of the array. Let is say we declare an array x in the following manner;

```
int x[ ]={10,20,30,40,50};
```

As you may have noticed this way of declaring array is different from that we used earlier. Here, while declaring the array, we have also assigned values to the elements.

So here "x" is an array and we have given 10,20,30,40,50 as the elements of the array.

That is x[0]=10,x[1]=20,x[2]=30,x[3]=40 and x[4]=50.

We will now define another integer pointer " j " using the statement int *j; and assign j=x; this statement basically mean "j" would point to the base address of an array "x".

Now we can read as the address of num[0] from j[0] and the address of num[1] from j[1] etc.. We see this in the program given below .

(pointer-4.c)

```
 #include<math.h>

#include<stdio.h>

         main()

        {

        int x[ ]={10,20,30,40,50};

        int i,*j;

        j=x;

        for(i=0;i<=4;i++)

         {

           printf("%u %u %d %d \n",&j,&j[i],*j,x[i]);

           j++;

         }

  }
```

This program implements the idea we just discussed to print out the addresses of the array elements. It also points out many features of pointers. Let us look at it in a little more detail.

After the usual include statements the main program has the array declaration we just discussed. Then we declare an integer i and integer pointer j. Then the assignment j=x, which make j point to the base address of the array x. Then we get into the loop which execute the printf statement five times by incrementing "i" one at a time. Each time it goes through the loop the pointer "j" is incremented by one.

The printf statement prints out the addresses of "j" and "j[i]". As we had discussed earlier the operator "&" will return the address of the variable or array following it. We are also printing out the value stored in the address to which "j" is pointing and the value of the array element "x[i]".

This will demonstrate the difference between the addresses of "j" and "j[i]. What we want to see here is that while the address given by "&j" remains the same the value given by "*j" changes, indicating that each time we increment "j" it is pointing to the next element of array "x". but the operator "&" acting on "j" always return its base address. So to get the address of the array elements we should use "&j[i]".

Note that we use % u as the format to print out the addresses and %d to print the value of an integer.

If we now compile and run the program using the commands,

```
gcc pointer-4.c

./a.out
```

the following lines will be printed on the screen;

```
 4277678728   4277678736  10   10

4277678728   4277678744   20   20

4277678728   4277678752   30   30

4277678728   4277678760   40   40

4277678728   4277678768  50   50
```

As we expected the first column is always printing the base address of j, even though we are incrementing "j" by "j++". But "&j[i]", in the second column, is printing different numbers i.e it is giving us the address of the different elements in array "x". So j[i] is pointing to the different elements of the array. The third column is "*j" and the fourth is "x[i]". You can see that there are the same. That is what we expect to find.

To summarize what we have seen here is that there are two ways to get the elements of an array. One can point an integer pointer to the array, then increment that integer pointer and pick up the values using the "" operator. Or one could use array elements straightaway. If you want to get the address of the array elements there is only one way of doing that. We the integer pointer is "j", printing &j we will not get the address of each elements. You could get only the base address while "&j[i]" will give us the addresses of each elements.*

Pointer to Pointers:

We can make a pointer to point to another pointer i.e *pointer to pointers*. It is useful when we want to read a character array and also a two dimensional array as we will see later.

Let us look at a sample code.

```
 char **marker;

char course[0]="numerical",course[1]="methods";

marker=&course[0];  \* pointer to course[0] which points to "numerical" */

marker++;                \* points to course[1] which points to "m"
in  "methods" */

(*marker)++              \* will then point to "e" */
```

Here "marker" is a pointer to a pointer, we use " ** marker " to declare this. In the sample code given above we also have character array "course" . The first element of this character array is "course [0]" which is storing the string "numerical" and the second element of the array is "course[1]" which is storing "methods".

In the third line of the sample code we point "marker" to "course[0]" . If we now increment "marker", by using "marker++", it will point to "course[1]".

Question:How do one read out each of the characters in a string ?

Here is a program that does it.

(pointer-5.c)

```
 #include<math.h>

#include<stdio.h>

        main()

        {

          int i;

          char **marker,*course[2];
```

```
   course[0]="numerical";

   course[1]="methods";

   marker=&course[0];

  for(i=0;i<=8;i++)

   {

    printf("%u %u %c \n",&marker[i],&course[i],*(*marker));

    (*marker)++ ;

   }

  }
```

This is the way we can do it. We have (*marker)++ which will move the pointer through "numerical", since "marker" is set to "course[0]". The bracket is very important here. Instead if we use "marker++" the pointer will shift to the next element of the array "course".

We have used "%c" to print each characters of the string to which (*marker) is pointing.

The printf statement also prints out the address to which marker is pointing and also the address of the array "course". You can compare these addresses.

Here is the out put of the program.

```
 4277434920  4277434920  n

4277434924  4277434924  u

4277434928  4277434928  m

4277434932  4277434932  e

4277434936  4277434936  r

4277434940  4277434940  i

4277434944  4277434944  c
```

```
4277434948  4277434948  a

4277434952  4277434952  l
```

What happens if you change the increment of the pointer to "course++" ?

Passing variables to functions:

A good programmer does most of the calculations in sub functions. When the "main" function (or other sub functions) invoke a particular sub function it may want to pass and receive some variables. In this section we will look at how this is carried out.

We have seen the use of a function earlier but haven't looked at how variables are passed.

(see below for general rules on calling a function). So now we will see how arrays and variables are passed from one function to another.

In this context the most important thing to remember is the following. *In C all variables are passed by value except arrays which are passed by reference*. What do one means by "by value" and by "reference" ?

There are two things a function can do, when you pass variable to it. The function can make a copy of the variable and work with that copy or it can directly work on the variable. In a C program the former method is adopted when we pass only the value of that variable to the function. That is the function gets a copy of the variable, stores it in a different location in the memory, and it works with that copy. So if you make a change in that variable value inside that function it doesnt change the variable in the master function. This is what we refer to as pass by value.

As opposed to this, in pass by reference , we pass the address of the memory location where the variable is stored. And now in the function if you change the value stored in that location it will of course change it in the main program too. So the main difference is that if you pass by ref it will not make a copy but if you pass by variable it will make a copy.

Passing an array to a function:

We do something quite different when we pass the content of an array from one function to another. This is because, as we had discussed earlier, the array name is also the pointer to the first element of the array. So when you are passing an array to another function, using the array name, you are passing pointer to the first element of the array, that is we are passing by reference.

In this case any, the function that received this array is directly working with it, not with a copy of it. So any change that is made in the array by the subfunction will also change the value of the array in the program from which its passed.

Summary:

All functions have the following form

type function name (arguments passed to the funtion)

variable declaration of the arguments

{

local variable declarations

statements

}

Argument passing: Two ways of passing variables are

(1) call by value

Here the function work with a copy of the variable

changes made in the function will not effect its value in

the calling function

(2) call by reference

Here the address of the variable is passed to the function.

changes made in the function will change its value in the calling function.

Let us now look at a sample program to see the difference between passing by value and by reference.

(pointer-6.c)

```
        #include<math.h>
          #include<stdio.h>
          main()
          {
          int num[ ]={10,20,30,40,50};
          int i,m,*j;
             m=5;
             j=&m;
   /*   Pass by reference */
             print(m,num,j);
           for(i=0;i<=4;i++)
          {
             printf("%u %d %d \n",&num[i],num[i],*j);
           }
   /*   Pass by value */
             print1(num[3],m);
              printf("%d %d\n",num[3],m);
          }
```

```
  print(n,b,l)
int n, b[5],*l;
{
int i;
b[4]=60;
*l=15;
for(i=0;i<=4;i++)
 {
{
   printf("%u %d %d \n",&b[i],b[i],*l);
 }
}

print1(int k,int n)
{
int i;
k=80;
n=20;
printf("%d %d\n",k,n);
}
}
```

An array "num", having dimension 5, is declared with the statement "int num[]={10,20,30,40,50};". Here the array elements takes the values

10,20,30,40 and 50. We also have an integer "m" and an integer pointer "j" declared. The pointer "j" is pointing the value of m. So we have an integer, and integer pointer and an integer array.

We then pass all the three into a function "print". Thus function receives "m" as "n", "num" as "b" and "j" as "l".

Inside the function "print" the last element of the array "b" is changed from 50 to 60 and "*l " is assigned a value 15. The address of the array elements "b" , the value stored in the array and the value of "*l" are then printed out. We will then return back to the main function.

After the call to "print", in the main function we print out the address of the array elements "num", the values stored in it and the value in "*j". We now see that the changes made in "b" and "*l" has changed "num" and "*j" as well. A look at the address of the array elements will tell us why !.

We saw the e.g of pass by reference. Now let see the second part of this program. Here we call another function "print1". In this we will be looking at an e.g for pass by value. By calling "print1(num[3],m)" we are passing the fourth element of the array "num" and the variable "m" to the function "print1". This function receives it as "print1(int k,int n)", that is the value in "num[3]" and "m" is *copied* to the integer "k" and "n" respectively.

I am also using this e.g to show you that the declarations of the variables that are received by a function can be in the function statement itself , as in "print1" or in a line immediately after the function statement, like in "print".

The values that the function "print1" receives for "k" and "n" are 40 and 15 respectively. Inside the function these values are changed to 80 and 20. Remember "k" was the fourth element of the array "num" and "n" the value of "m". We print these values inside this function and compare it with the values of "num[4]" and "m" printed in the main function. We see that the changes in the variable values we made in the "print1" function has not changed the variable values in the main.

This demonstrates the difference between passing by reference and passing by value.

The next thing we will look at is two dimensional arrays and connection to pointers to pointer.

Some examples of Arrays and pointers

In the last lecture we saw about pointers and arrays, pointer to a pointer and pointer to arrays. We will summarize some of these concepts through the example below

Pointer-Summary-1.c

```
#include <stdio.h>
#include <math.h>
main()
{
	int num [ ] = { 10,20,30,40,50 }
	print ( &num, 5, num);
}
print ( int *j, int n, int b[5])
{
	int i;
	for(i=0;i<=4;i++)
	{
		printf (  "  %u  %d  %d  %u  \n  ", &j[i]  ,  *j , *(b+i)  ,  &b);
		j++;
	}
}
```

In this example we have a single dimensional array num and a function print . We are passing, the address to the first element of the array, the number of elements and the array itself, to this function. When the function receives this arguments, it maps the first one to another pointer j and the array num is copied into another array b . (The type declarations are made here itself. Note that these declarations can also be given just below this line). j is now a pointer to the array b.

Inside the function we are printing out the address of the array element and the value of the array element in two ways. One using the pointer j and the other using the array b. If we compile and run this code we get the following out put,

```
3221223408 10 10 3221223376
3221223416 20 20 3221223376
3221223424 30 30 3221223376
3221223432 40 40 3221223376
3221223440 50 50 3221223376
```

Note that as we increment j it points to the successive elements of the array. We can get both the address of the array elements and the value stored there using this. However the array name, which acts also as the pointer to its base address is not able to give us the address of its elements. Or in other words, the array name is a constant pointer. Also note that while j is points to the elements of the array num, b is pointing to its copy.

Next we have an example that uses a two dimensional array. Here care should be taken to declare the number of columns correctly.

Pointer-summary-2.c

```
#include <stdio.h>
#include <math.h>
main()
{
           int arr [ ][3] = {{11,12,13},
{21,22,23},{31,32,33},{41,42,43},{51,52,53}};
           int I , j ;
        int  *p ,  (*q) [3], *r ;
        p = (int *) arr ;
        q = arr;
        r = (int *) q ;
        printf    ( "  %u  %u  %d  %d  %d  %d    \n  ",    p    , q , *p    ,
*(r)  ,  *(r+1),  *(r+2));
        p++ ;
        q++ ;
        r = (int *) q ;
        printf    ( "  %u  %u  %d  %d  %d  %d    \n  ",    p    , q , *p    ,
*(r)  ,  *(r+1),  *(r+2));


}
```

Here we have a pointer p and a pointer array q. The first assignment statement is to make the pointer p points to the array arr. While assigning, we also declare the type of the variable arr. Note that variables on both side of this statement should have the same type. Next line is a similar statement , now with q and arr . Since q is a pointer array, the array can be directly assigned to it and there is no need for specifying the type of the variable. In the next line we make the pointer r to point to the pointer array q . Then we will print out the different values. Here is what we get from this ,

```
        3221223344  3221223344  11 11 12 13
        3221223348  3221223356  12 21 22 23
```

Here we see that incrementing p make it just jump through each element of the array, where as incrementing q, will move it from one row to another row.

Passing 2 Dimensional Arrays to a Function

So far we have looked at the ways to pass a one dimensional array to a function. Let us now extend this to the case of a two dimensional array. There are four different ways of passing a two dimensional array to a function

- An array to an array a[m][n] à b[m][n]
- An array to a pointer a[m][n] à * p
- An array to a one dimensional pointer array a[m][n] à (*q)[n]
- An array to a two dimensional pointer array a[m][n] à (*q)[m][n]

We will illustrate the use of these four methods through the sample program. In this we pass a two dimensional array to four functions, using each one of the methods mentioned above.

two-dim-array.c

```
#include<math.h>
#include<stdio.h>
main()
{
        int arr[ ][2]={{11,12},{21,22},{31,32},{41,42},{51,52}};
        int i,m,n,*j;
         m=5;
         n=2;

         print(arr,m,n);
        show(arr,m,n);
        display(arr,m);
        exhibit(arr,m);
}

Print (int b[ ][2],int l,int p)
{
         int i,j;
```

```
            printf("print \n");
            for(i=0;i<l;i++)
            {
                        printf("%d %d \n",b[i][0],b[i][1]);
            }
}


Show (int *b,int l,int p)
{
            int i,j;
            printf("show \n");
            for(i=0;i<l;i++)
            {
                        printf("%d %d \n",*(b+i*2+0),*(b+i*2+1));
             }
 }
display (int(*b)[2],int l)
{
           int i,j,*p;
           printf("display \n");
           for(i=0;i<l;i++)
            {
                         p=(int*)b;
                         printf("%d %d \n",*(p),*(p+1));
                        b++;
            }
}

exhibit (int (*b)[5][2],int l)
{
           int i,j;
           printf("exhibit \n");
           for(i=0;i<l;i++)
            {
                        printf("%d %d \n",(*b)[i][0],(*b)[i][1]);
             }
}
```

The functions, **print, show, display** and **exhibit,**are used to demonstrate how the arrays are passed as arguments, the function call is the same but the way the arguments are received in the function definition is different. **Print** gets it as an array, **display** gets it as a one dimensional pointer array, **show** gets it as a pointer and **exhibit** as a two dimensional pointer array. All the functions do exactly the same thing, that is to print out the elements of the array. Note the different ways the elements are accessed inside each of these functions.

In the function **show** the two dimensional array is received as pointer. On incrementing this pointer we can access each of the elements of the two-dimensional array . The elements of the array are read row by row.

In the function **display** the two dimensional array is received as a one dimensional pointer array b. By default this pointer array, which had the same number of elements as the number of columns in out two dimensional array, is pointing to the first row of the two dimensional array. To access this elements we now need another pointer, which is defined here as p . We then make the assignment p=(int*)b inside this function and read the two elements of the first row. When we increment b by b++ we go to the next row and so on .

In the case of function **exhibit** each element of the two dimensional array of pointers is pointing to the corresponding element of array. Please note the use of paranthesis to read the elements of this array of pointers.

Returning Array from a Function

While there are four different way to pass a two dimensional array to a function , there are only three different ways to return a two dimensional array from a function. They are,

- array to a pointer   a[m][n] → *p
- **array to a one dimensional pointer array   a[m][n] → (*q)[n]**
- **array to a two dimensional pointer array   a[m][n] → (*q)[m][n]**

As in the earlier case we will look at a sample program to understand this better.

Return-array.c

```
#include<math.h>
#include<stdio.h>
#define l 4
main( )
{
        int *a,i;
        int *matrix1( );
        int (*b)[2];
        int *r;
```

```
          int (*matrix2( ))[2];
          int (*c)[5][2];
          int (*matrix3( ))[5][2];

           a=matrix1( );
            printf("matrix1 \n");
           for(i=0;i<l;i++)
                      printf("%d %d \n", *(a+i*2),*(a+i*2+1));
          b=matrix2( );
           printf("matrix2 \n");
           for(i=0;i<l;i++)
           {
                      r=(int*)b;
                      r=(int*)b;
                      printf("%d %d \n",*r,*(r+1));
                      b++;
            }

          c=matrix3( );
          printf("matrix3 \n");
          for(i=0;i<l;i++)
           {
                      printf("%d %d \n",  (*c)[i][0],(*c)[i][1]);
           }
}


int * matrix1( )
 {
           static int arr[ ][2]={{11,12},{21,22},{31,32},{41,42},{51,52}};
           return *arr;
 }

 int (*matrix2( ))[2]
 {
           static int arr[ ][2]={{11,12},{21,22},{31,32},{41,42},{51,52}};
           return arr;
 }


 int (*matrix3( ))[5][2]
 {
           static int arr[ ][2]={{11,12},{21,22},{31,32},{41,42},{51,52}};
           return (int(*)[5][2])arr;
```

```
}
```

Here in the function matrix1, which is declared as a pointer, we have a two dimensional array with 5 rows and 2 columns and it returns this array as a pointer. To receive it the main function should have an integer pointer. That is in the statement a=matrix1( ); in the main function a should an integer pointer. What is returned here is the address to the array arr.

In the function matrix2, which is declared as a one dimensional array of pointers, the return statement is quite different from the other two. Here array is returned by simply its name. In the main program the statement

b=matrix2( ); receive it through the one dimensional pointer b.

In matrix3 which is declared as a two dimensional array of pointers, the return statement should specify the full dimension of the array and its variable type. It is received in the main program by a two dimensional pointer array.

In all the cases with pointers and arrays we have looked at so far, the array dimension was fixed. However there are cases in which we would like to change the dimension of the arrays dynamically. We will look at this aspect of pointes in the next lecture.

Dynamic Memory Allocation

When we declare an array, we need to reserve some memory to store the elements of this array. This memory allocation and it is static. That is when we declare an array, we specify the number of elements in that array and a fixed memory is allocated. Once declared the size of the array cannot be changed.

But there are situations in which we may want to declare the variable and not allocate memory for it until we use that variable. We may also want to free the memory allocated for a variable after its use is over. These are some of the things we will concentrate on in this lecture.

The dynamic allocation of memory during the program execution is achieved through two built in functions *malloc* or *calloc, realloc and free*. There is also *sizeof()* function used to determine the number of bytes occupied by an entity in memory. Let is look at the use of these functions through a couple of examples.

Dynamic_memory.c

```
#include<math.h>
#include<stdio.h>
#define l 4
main()
{
            int *a, i,*p;
           float *b,*c;
           a=(int*) malloc(l);
           for (i=0; i<l; i++)
          {
                      *(a+i)=i;
         }
           for (i=0; i<l; i++)
           {
                       printf ("a %d %d %u \n",i,*(a+i),(a+i));
           }
            p=(int*) calloc(l,4);
            for (i=0; i<l; i++)
         {
                 *(p+i)=i*2;
         }
           for (i=0; i<l; i++)
           {
                      printf(" p %d %d %u %d\n",i,*(p+i),(p+i),sizeof(p));
           }
         b=(float*)malloc(l);
          for(i=0;i<l+2;i++)
         {
                 *(b+i)=i*4.0;
         }
           for(i=0;i<l+2;i++)
           {
                       printf(" b %d %f %u %d\n", i,*(b+i),(b+i),sizeof(b));
           }
          c= calloc(l,8);
```

```
        for(i=0;i<l;i++)
         {
               *(c+i)=i*4.0;
        }
        for(i=0;i<l+1;i++)
        {
                    printf(" c %d %f %u %d \n", i,*(c+i),(c+i),sizeof(c));
        }
         for(i=l+1;i<l+2;i++)
        {
          printf(" b %d %f %u \n", i,*(b+i),(b+i));
        }
}
```

In the program above we create an integer pointer ***a** **and** two floating point pointers ***b,*c.** Now we want to store a string of numbers using these pointers, instead of using a fixed size array.

To achieve this we fist have to allocate enough space in the memory to which the pointers point to. We will allocate just enough memory to store the elements we want to . For example in the above program we want to store 4 integer values. By using the malloc function we will allocate memory for an integer array of size 4. Note the usage of malloc () here. We invoke it by the statement a=(int*) malloc(l); We could also do the same with p=(int*) malloc(l*sizeof(int)); or p=(int*) malloc(l*size(a)); where l=4. One important thing to note here is that since malloc returns a **void** pointer, we need to type cast it before equating it to the pointer we want. In the case above **a** is an integer so we have to use (int *) before malloc. Similarly we see the use of a malloc to allocate memory for a float pointer **b** . We use the pointer p to demonstrate that we can do exactly the same using the function calloc. The function is exactly the same , except that we need not type cast it now and also that we have to give a second argument to **calloc**,which is the number of bytes need to store one variable. In most of the modern compiler this value is ignored by the function.

So far we have seen operations which is pretty much like that in the fixed size arrays. The only difference is that we assign the memory only just before the variable is used. This by itself is not very useful unless we can change this memory allocation , that is increase it if we want more and remove it if we don't want to use the variable again in the program. This is what **realloc** and **free** will do for us. Let us look at that with an example.

```
realloc.c
/*      Dynamic memory allocation realloc and free */
        #include<math.h>
        #include<stdio.h>
        #define l 5
        main()
        {
        float *a;
        int i;

        a=(float*) malloc(l);
        for(i=0;i<l;i++)
            { *(a+i)=i*3.0;}
        for(i=0;i<l;i++)
        {
         printf("a %d %f %u \n",
                   i,*(a+i),(a+i));
        }
       a=(float*)realloc(a,sizeof(float)*(l+3));
        for(i=0;i<l+3;i++)
            { *(a+i)=i*3.0;}
        for(i=0;i<l+3;i++)
        {
         printf("a %d %f %u \n",
                   i,*(a+i),(a+i));
        }

        free(a);
        }
```

In the above program we have a pointer a which is of type float . We then give it enough momory space to store 5 elements. After filling and printing out the address of those locations and the value stored in them we increase the"size" of the memory allocated by the statement "a=(float*)realloc(a,sizeof(float)*(l+3));". This statement will increase the memory space to store 3 more floating points. You can see by yourself that if you remove this line and try to store more than5 elements into a you will get segmentation fault. The last statement in the program "free(a);" free all the space allotted to a .

Thus by using malloc, realloc and free we can change the memory taken up by the program as required by it. This is particularly useful when we have to use large temporary arrays in a calculation.

Functions and pointers to functions.

We saw how to pass array to a function, return array from a function, and declare arrays, pointer arrays, pointers to arrays and also dynamic memory allocation. We will no see a different use of pointers, that is to invoke function calls. Before we go on to the the description of pointer to a function, lets have quick review of the structure of functions

All functions have the following components.

- **Definition**
- **type function name (arguments passed to the function)**
- **variable declaration of the arguments**

{ local variable declarations;
    statements;}

- Argument passing

Two ways of passing variables to a function are

(1) call by value

Here the function work with a copy of the variable. Changes made in the function will not effect its value in the calling function

(2) call by reference

Here the address of the variable is passed to the function. Changes made in the function will change its value in the calling function.

Let us look at an example to remind ourself about these differences.

- **Example (variable-passing.c)**

```
#include<math.h>
#include<stdio.h>
#define l 4
main()
{
        float b; int a[2],*c;
        void myfunction();
        c=(int*)malloc(1);
        b=10.0;
```

```
           *c=5;
            a[0]=1;a[1]=2;
            printf ("before function  call  %f %d  %d %d\n", b,a[1],a[2],*c);
           myfunction (b,a,c);
           printf ("after  function   call  %f %d %d %d\n", b,a[1],a[2],*c);
 }

void myfunction (x,y,d)
 float x;
 int y[2],*d;
{
         float z;
         x=2*x;
         y[0]=3*y[0];
         y[1]=3*y[1];
         *d=*d+2;
 }
```

In this example the main function is passing "b,a and c" to "myfunction", which receives it as "x,y and d ". Since a is and array and c is a pointer they are passed by "reference" while b is passed by value. Inside myfunction all the values are altered. We will printout the values of "b,a and c" in the main function before and after the call to myfunction. This is what we get,

```
before function  call  10.000000 1 2 5
after    function  call  10.000000 3 6 7
```

We see that the variable that is passed by value does not get altered while any changes made to quantities, that are passed by reference, in the sub function will alter its value in the main function as well.

Pointers to Functions

On may occasions we may want to write a program to which a user defined function can be passed. For example , as you will see later in this course, one can write a general program to solve an ordinary differential equation. We would then like to have it in such a way that a user can write a function, which evaluate all the derivatives of the particular problem, and pass it to the ODE solver. We will now see how this can be achieved by using a pointer.

Function name can be used as a pointer to point to functions like the name of the array is a pointer to its base address. We can also define a pointer, point it to a function and then use that pointer to invoke the function. The pointer to a function should be of the same "type" as the function.

If we have to pass a function pointer to another function, we have to define them as "external". That is they are "global" to the program and is defined outside any function boundaries. Lets look at an example for this now**.**

```
/*Pointer to a function and passing the pointer to a function */
  passing-function.c
/*      Pointer to a function and passing the pointer to a function */
        #include<math.h>
        #include<stdio.h>
        #define l 4
       float cube(float);
        main()
        {
        float b;
        void (*func_ptr)();
        void myfunction();
        b=2.0;
        func_ptr=myfunction;
        (*func_ptr)(b,&cube);
        }


        void myfunction (x,powerthree)
         float x;
         float (*powerthree)(float);
       {
          float z;
          x=2*x;
          z=(*powerthree)(x);
           printf("%f\n", z);
}

  float cube(x)
   float x;
  {
    float z;
    z=x*x*x;
   return z;
  }
```

Here cube is an external function and is defined outside of the function boundaries. This function calculates the 3rd power of any floating point variable passed to it and returns it as a floating point. Thus this function is of the type "float".

In the main function we have defined a pointer func_ptr. Since we want to use it to a function which is of the type "void" we need to define this pointer also as "void".

The function myfunction has a floating point and a function pointer as its arguments. Instead of invoking this function directly, we use func_ptr for it. Note that we are passing the address of the external function, cube, as an argument here. myfunction receives it as a floating point x and function powerthree. When powerthree is called inside this function, it is actually invoking the function cube, which is supplied as external.

We thus see the use of function pointer and also the way to pass a function to another function here. We will make use of them later in the coarse while solving ordinary differential equations.