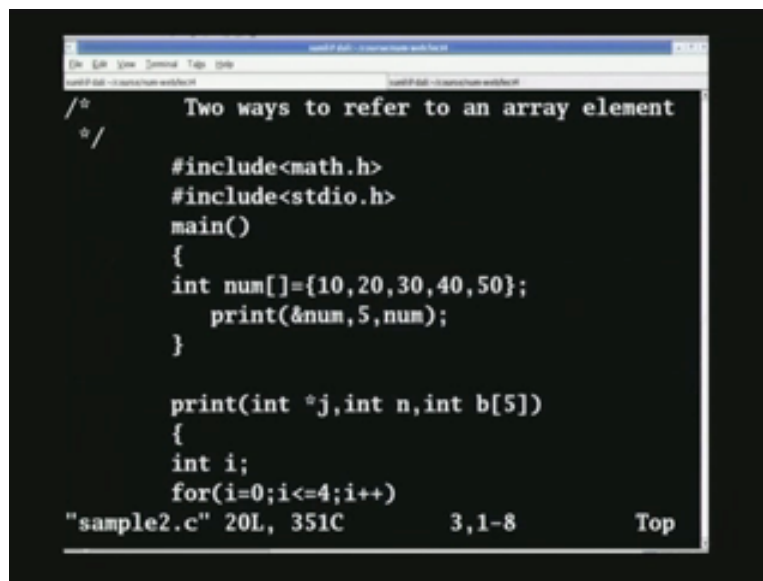


Numerical Methods and Programming
P. B. Sunil Kumar
Department of Physics
Indian Institute of Technology, Madras
Lecture - 4

Programming - External Functions and Argument Passing

In the last lecture, we saw the use of pointers, arrays, and pointing a pointer to another pointer or a pointer to an array. So we will summarize that by looking into this program.

(Refer Slide Time: 01:34)



```
/*      Two ways to refer to an array element
*/
#include<math.h>
#include<stdio.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("%d\t",b[i]);
}

"sample2.c" 20L, 351C      3,1-8      Top
```

So here is a program which we would make use of many of these, so a one-dimensional array. So let us go through this program. So we have declared an array here okay, and then we are calling a function called print okay, and the print passes this pointer to this array. That is, we are passing the name of the array, which is a pointer to the first element of the array, into this function, and then we are passing the dimension of the array, which is 5 and then we are passing the array itself.

We are doing 3, so this is basically to summarize what we did in the last lecture. So thus we are passing the address, to the first element of the array and we are passing the array itself to the function. What will the function do? The function maps this address to another pointer, j. So that is, we want to show how to point a pointer to an array. So here is this address is being passed on to another pointer and this is being passed on to another integer, n, and then this num, which is an array, is passed on to another array b. Again, this type of declaration here can either be inside this function call itself or it can be given just below that.

Okay so then what does this do? This prints out the address. Now we are going through a loop here and then we are printing out the address of the array elements in two different ways. So we are going to print out the address of the array element here

by using this pointer `j` okay, and then we will print out the address of the array `b`. So we will see that. This will be always pointing to the base address, and this will be running through that okay, and to get the element of the array itself we can either use asterisk `j` or asterisk `b` plus `i`. These are the two different ways of getting the array elements. So that is what we will see if you just compile this and run it once again. So you can see that this is always pointing to one address which is the base address of the array and this is running through the array and we get by referring to the array elements as either asterisk `j` or asterisk `b` plus `i`. We get the same answer. That is something which we saw yesterday.

So another one which you should look at is again pointing a two-dimensional array which was what we saw yesterday. Now how do we point a two-dimensional pointer to a two-dimensional array? There are 2 different ways of doing that again. We could either point a pointer to that array or we could point a pointer array to the array to the two-dimensional array. Okay so that is again demonstrated in this program which we saw yesterday. We will go through this once again. That is, we have declared a two-dimensional array here. So it has 3 columns and 5 rows. That is the way we declare okay. Again, I said when you are declaring a two-dimensional array we should always be careful that we should always declare the number of columns. The number of rows are not important, but the number of columns are important.

Okay so now we have a pointer here. There are 3 different pointers, two pointers `p` and `r`, and we have pointed array `q`, all of type integer. Now we are going to point pointer `p` to the array, so that, always when you do this, you have to typecast it as I mentioned in the last lecture, again. So you have to typecast it. It is an integer type and an array. So we are pointing now this is the pointer array which you are pointing to the two-dimensional array, `arr`. So they are the 3 different ways of doing this. Now we are using pointing this pointer `r` again to point to this pointer array. So you can notice that again you have to typecast it. So this is a pointer, and this is a pointer array. Here is a pointer and this is again array. So then we will print out all these quantities. We will print out what is `p`, what is `q`, what is asterisk `p`, asterisk `r`, `r` plus 1, `r` plus 2, etcetera.

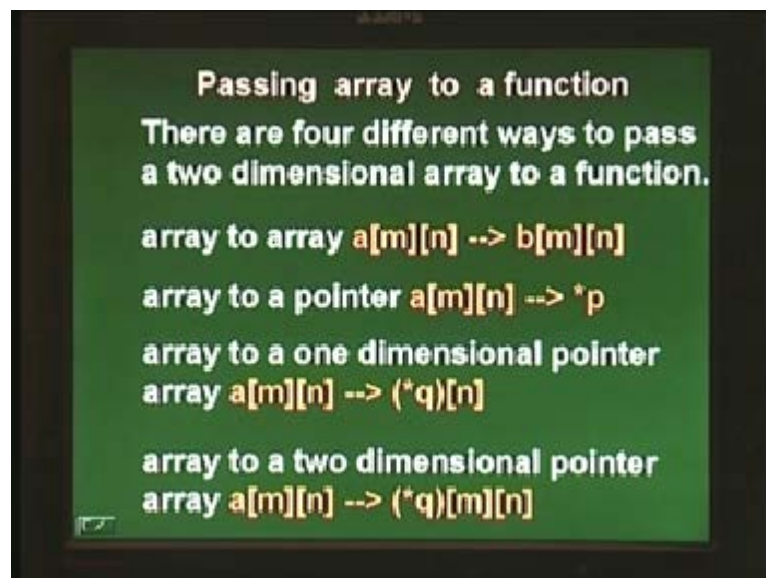
So remember, what we said was if you increment `p` here, we increment `p`, it will simply run through the array, while if you increment `q` it will jump from, it will go from row to row. So if this `q` which is pointing to the first row by this definition, first row first element okay, now we have this pointer pointing to that array and then by saying `r` plus 1, `r` plus 2, etcetera, we will get the different elements of that row. So if you increment `p` here, it will go to the next element of the array, while if you increment `q` here, it will go to the next row of the array. So that is what we have to remember, and then every row we read off by pointing `r` to that array and then we run this. So what we do is, we will first point everything to the base addresses and then print out that and we will increment `p` and `q` and again we print out that. So that is what we see.

So here we have base addresses `p` and `q`. Both are pointing to the same address and we are running through the array. So we have 11. So `p` is pointing to 11 and we are running through the whole row and we get 11, 12, and 13. And then we increment `p` and we increment `q` and when we increment `p`, which was a pointer, it goes to the next element of the array while we incremented `q`, it went to the next row of the array, and

we have 21, 22, 23 as the elements of that row. So that is what we saw in the last lecture. We have to now, looking at it, how to pass a two-dimensional array into a function. We saw how to read off two-dimensional arrays and how we can point pointers to that and point arrays to that, etcetera. Now we will see how to pass a two-dimensional array into a function.

So we said that there are four different ways of doing that. One is to pass an array to another array in the function. That is here used as array a m dimension m and n, and that is passed on to another array b m and n. So we could do it in another way. That is, we could pass an array into a pointer, p. Okay so now, p is just pointing to a m n. So we could increment p by 1 and read off all the elements of the array one by one in a serial fashion. And another way to do is to point a one-dimensional array or we could pass the elements of the array into a one-dimensional array of the pointer. So here it was passed to another array and here it passed to a pointer, and here it is passed to a pointer array, a one-dimensional pointer array or we could do that by just passing to a two-dimensional pointer array.

(Refer Slide Time: 09:23)



So there are 4 different ways of passing an array to a function. So I repeat that an array to an array, an array to a pointer, and an array to a one-dimensional array and an array to a two-dimensional array. So that is what we should see in a sample code. So here is the sample code for that. We have declared an array, so this array again has 2 columns and 5 rows and then we have declared here 3 integers and 1 pointer and the array dimensions are 5 and 2; 2 columns and 5 rows. And now we would call 4 different functions. So we call four different functions and pass the same thing. The call is the same. We are passing array, the dimensions of the array, the number of rows and number of columns to all these functions. We call print, show, display etcetera. So 4 different functions.

(Refer Slide Time: 10:07)

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

(Refer Slide Time: 10:09)

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

So now, what are these functions? These functions are given here. So the print gets it as an array, a two-dimensional array. So it gets an array. Display gets it as one-dimensional pointer array, show gets it as pointer and exhibit gets it as a two-dimensional pointer array. These are the 4 different ways of doing it. So first, it is just array to array. That is, remember the previous call, that the call is the same. Everything is just passing the array but in the function the print is as a two-dimensional array and in the program, the function called show gets it as a one-dimensional, gets it as a pointer, and display gets it as a one-dimensional pointer array, and exhibit gets it as a two-dimensional pointer array.

Now you see that we have declared that here. So we got as integer here, declared integer pointer here and integer pointer array here and integer two-dimensional pointer array here. So how do we print them out? We can here pass it to a function as

array itself. We can print the elements of the array simply by saying, by just reading out the array elements. When we pass it as an array. Here, it is a pointer. So here the array is passed to a pointer. Now we have to increment the pointer and run through the whole array. That is what we are doing here. You can watch. We need this number 2 here. That is the number of columns which we have, okay so i basically is running through all the rows. I could have put in here p straightaway. Instead of i star 2, we could have put i star p because p is 2, which is the number of columns which we passed.

So now we were running through this, all the rows and printing out both the columns. So the first row that is i is 0. So that is the first row. Okay we print the first element, and the second element and then we increase I and then go into the first element, of the second row, and next element. So that way we can print out all of them.

(Refer Slide Time: 12:33)

```
Sample5.c
/*passing array to a function */
#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;
```

(Refer Slide Time: 12:44)

```
print(arr,m,n); /* array to array */
show(arr,m,n); /* array to pointer */
display(arr,m); /* array to one dim.
                pointer array */
exhibit(arr,m); /* array to two
                dimensional
                pointer array */
}
```

(Refer Slide Time: 12:56)

```
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]);
    }
}
```

(Refer Slide Time: 13:08)

```
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));
    }
}
```

Okay so that is what we are printing out. So let us see that one by one. So the print just prints out all the elements. So it is 1, 1, 1, 2, 2, 1, 2, 2, 3, 1, 3, 2, 4, 1, 4, 2, 5, 1, 5, 2. So it is just printing out all the elements like this okay. Now here we are using this asterisk sign to read out the number of the array because b is a pointer. We increment b by this element and by this element and by this, the number of rows and columns, and then we print that out, and we get exactly the same numbers, of course. And then the third way is to use the display function which is actually getting the array as a one-dimensional pointer, one-dimensional pointer array okay. So now, one-dimensional pointer array we get.

(Refer Slide Time: 13:21)

```
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++;
    }
}
```

(Refer slide time: 13:34)

```
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]);
    }
}
```

We need another pointer to point to that and read out the elements, and that is what we do here. We have declared another pointer p and p is now pointing into the array, b, the pointer array, b. So now, p is a pointer and this is a pointer array. So we have to typecast it okay so that p equal to integer star b. Then we increment p. We take p and p plus 1. That gives you the array. Then we increment b. That goes to the next row. So then, we again point p to the incremented b, and again print out that. That is what we are doing in the display function. So we get 11, 12, 21, 22, 31, 32, 41, 42, 51 and 52.

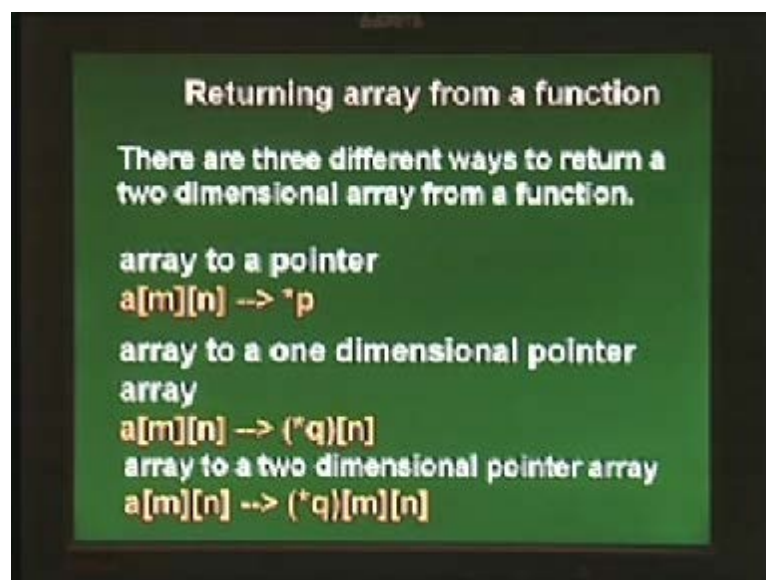
So we have got array to be printed by using a by passing it on to a function which is a one-dimensional pointer array and the next one is to pass it on to a two-dimensional pointer array. Okay so that is declared here. Now b here is a two-dimensional pointer array with array dimensions 5 and 2. So now we can read out this simply by this

statement. So that is very similar to using this array itself, very similar to that. Only thing is we have to use the asterisk sign to print out the actual variable value.

So that is what we are doing here. So we printed out by going through all the rows and the 2 elements of the row, or the two columns. We just print that out and that we can see here again. Again, we get the same numbers. So what we have done, we have seen two different ways, 4 different ways, of passing an array to a function. So that is array to array, array to a pointer, array to a one-dimensional pointer and an array to a two-dimensional pointer. So that is what we have done. Okay so now, we have to see how the function can return an array to the main program. That is what we are going to see next. Now returning array to a function. That is what we are going to see in the next one. We saw 4 different ways of passing an array to a function. Okay now what we see is that the way we can return it, there are 3 different ways.

So that is again array to a pointer. The program is returning the function, is returning an array after doing whatever it has to do with that, back to the main program or the program from which it is called. So the main function from which it is called. One way to do that is to return an array to a pointer or we could return an array to a pointer array, or we could return an array to a two-dimensional pointer array. Okay that is again three different ways. We cannot return an array to an array. You might have noticed we have three different ways of doing that. Array to a pointer, an array to a pointer array, and an array to a two-dimensional pointer array. Again, the best way to see this is to look at an example. So here is an example, a program.

(Refer Slide time: 17:40)



So here it is, the main program, and then you have the functions. We are calling the functions matrix 1, matrix 2 and matrix 3. We will see this later. This function calls for a little more detail. So here we are declaring the functions. The function has to return something. The function is going to return an array. That is what this program is going to demonstrate. Because of that we have to say what type of the variable the function is going to return. It is an integer. The function is declared here. It is an integer. So matrix 1, matrix 2 and matrix 3 are integer pointers. Now, here is our

pointer array declared, and here is another pointer declared, a and r are two pointers declared, and i is an integer. So now we will see.

(Refer Slide Time: 17:50)

```

#include<math.h>
#include<stdio.h>
#define I 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<I;i++)
{
printf("%d %d\n",
*(a+i*2),*(a+i*2+1));
}

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

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

```

So we have a matrix a, sorry, pointer a, that is been pointing to matrix 1, this function. What is matrix 1 doing? We will see that later. So this matrix 1 and we are going to print those values. So that is what we are going to see. We will see this program, and we will go through the details. Returning array from a function that is what we are going to see. So this is the same program that is listed there. We have matrix 1 and matrix 2 and matrix 3 as that. We are going to run this program and before we run the program, we will just look at the function matrix 1.

(Refer Slide Time: 19:23)

```

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;
}

```

So here is the function matrix 1. Okay we will look at them one by one. So we will look at function matrix 1 that has an array declared there. So you have an array

declared. That is, array arr, and that has got 1, 2, 3, 4, 5 rows and 2 columns, okay and it returns this array. So what does this function do? The function simply returns this array, an array is declared here and it is returned. The array is returned as a pointer, and remember the program above, I said a, which is a pointer that is equal to, that is my statement. So a equal to matrix 1. So matrix 1 would simply return the array into this pointer, a. That is what happens, and then what happens when you run this? We will see that. Okay so we have matrix 1. I am printing out this for clarity. So this is matrix 1 and then we are printing out the array elements.

So we print out only four array elements here okay because I said i less i less than 1. So it goes up to 0, to 3. So there are four rows I am printing out. That is, 1, 1, 1, 2, 2, 1, 2, 2 3, 1, 3, 2, 4, 1, 4, 2. So you can notice here I am running through this pointer, a is a pointer. So I am running through, as we did in the case when you pass the function to the array, an array to the function. We are doing exactly the same thing here. Now we are printing in the main function, not in the, not in the main program, not on the function. So we are printing out that.

So this a is equated to matrix 1 and we saw matrix 1 simply returns an array to this, to this pointer. So that is what we did again. So let us look at that again. So it declares an array, and that array was returned as a pointer to the main program. Remember, I have to declare the array as a, a as a pointer here and also matrix 1 as a pointer here, with a type integer. This is important, because it is returned to that. Next one is matrix 2. Matrix 2 is declared as type integer. It is a function and it is a pointer array. This function is declared as a pointer array.

Please notice the way it is declared. It is a pointer and it is an array. It is an array of 2 columns. That is the way it is declared. Then what does it do here? The matrix 2 part, it is again, it is declared as an array very similar to this. It is declared as an array, and it is returned now as, notice the difference, it is returned as asterisk arr here and it is returning as arr here. That is what we see here also. So we have matrix 2 and the matrix 2 is simply returned as arr. I am going to when you run that you will see that you get exactly the same value. This is another way of doing the same thing, but it gives you exactly the same value as matrix 2 gives you exactly the same value as the previous one.

So now we have the third call that is now we saw returning into a pointer and we saw returning to a pointer array, one-dimensional pointer array okay and there are different ways of returning this. We have to remember that. That is, even though the call is the same, the way it is returned is different. So there is an asterisk sign arr, here is simply arr. Now we see how we can return a matrix into a two-dimensional pointer array. Okay so now again, the return statement is very very different. We have to put a type statement here, and then return that. We said integer star, the dimension of the array and the array.

So the same array is declared here and we return this in this fashion, and we get as, you can see we get exactly the same values. Okay so let us go back and once again look at the calls. So we had the first function matrix 1 declared as an integer pointer, the second function matrix 2 declared as an integer pointer array and the third function matrix 3 which is declared as an integer two-dimensional pointer array okay. Notice that this is a two-dimensional pointer array and so depending upon what the

function is returning, we have to declare it properly, to what value it is declaring, what is the variable or pointer to which it's declaring, that also has to be declared properly.

Okay so here we return it to a, we say a is equal to matrix 1, so that has to be the same type. So we have an integer and a pointer. This is the pointer, and this matrix 2 returns that into a one-dimensional pointer array. So we have declared both b as a one-dimensional pointer array, and matrix 2 as a one-dimensional pointer array. So that is one thing. Okay and matrix 3 is returning the function into a two-dimensional pointer array. This c again, we have to declare c as a two-dimensional pointer array and matrix 3 as a two-dimensional pointer array. Okay so that is the summary.

(Refer Slide Time: 25:06)

```
Sample6.c
/*returning array from a function */
#include<math.h>
#include<stdio.h>
#define l 4
main()
{
int *a,i;
int *matrix1();
int (*b)[2];
```

(Refer Slide Time: 25:17)

```
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));
}
```

(Refer Slide Time: 25:28)

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

(Refer Slide Time: 25:39)

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

The summary is that we have 3 different ways of returning an array, a two-dimensional array into function, and we have 4 different ways of passing a two-dimensional array into a function. So that is what we saw. So now, before we go into other details about function, one more point about the pointers which we should know is the dynamic memory allocation. This is an important fact that when we declare arrays, we saw arrays so far, so when we have arrays, we have, when you declare an array, we allocate some memory to it and that memory allocation is static. The moment you declare an array, that memory has been allocated to it. But sometimes, we want to declare something but we do not want to allocate the memory till we do something else. We actually use that variable. So in that case, which is useful to have a way of doing this. So that is one way of doing that is it uses dynamic memory allocation okay.

(Refer Slide Time: 25:50)

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

(Refer Slide Time: 26:02)

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

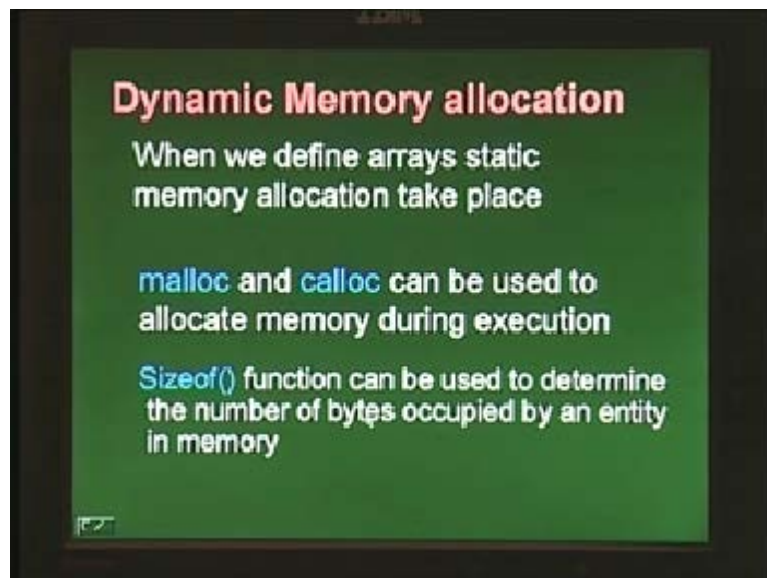
So that is the use of this again I may emphasize there are two, the reason for, or the use of this function is that when we define arrays it uses a static memory that is, memory allocation is static. It takes place in the beginning of the program, but we do not, we may not need this all the time or we may want to have memory allocated to the allocated only when we use that variable and then we can use the functions called malloc and calloc. There are two built-in functions called malloc and calloc and we can use that to allocate memory just during execution. So that is what we are going to see now.

(Refer Slide Time: 26:13)

```
int (*matrix3())[5][2]
{
    static int arr[3][2]={{11,12},{21,22},
                          {31,32},{41,42},{51,52}};
    return (int(*)[5][2])arr;
}
```

So and then, we can use the size of function, so that is another function. So there are three important functions with dynamic memory allocation; one is malloc calloc, which is actually similar, in fact, almost the same in most of the compilers okay, and then another thing called sizeof. This is to allocate memory, and sizeof can be used. It is a function which will be used to determine the number of bytes occupied by an entity which we have allocated in the memory too. That is what we would see now okay.

(Refer Slide Time: 28:29)



Dynamic Memory allocation

- When we define arrays static memory allocation take place
- malloc and calloc can be used to allocate memory during execution
- sizeof() function can be used to determine the number of bytes occupied by an entity in memory

So here is a program which uses all of that. So let us go through this program and then we will run it, and then see step by step. So we have declared an integer pointer a and pointer p and an integer I and then we use float pointers b and c okay. So now, we want to put in a value, some number, into this a. So you cannot simply say a equal to

star a equal to something to begin with, because a is just declared as a pointer, but we have not allocated any memory to that okay. So it is just an address right now.

(Refer Slide Time: 28:33)

```

#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 ("%d %d %d %u\n", i, *(a+i), *(a+i));
    }
    p=(int*) calloc(l,2);
    for (i=0; i<l; i++)
    { *(p+i)=*2;}
    for (i=0; i<l; i++)
    {
        printf ("%d %d %d %u %d\n",
            i, *(p+i), *(p+i), sizeof(p));
    }
}

```

```

b=malloc(l);
for(i=0;i<l;i++)
{ *(b+i)=*4.0;}
for(i=0;i<l;i++)
{
    printf("%d %d %f %u %d\n",
        i, *(b+i), *(b+i), sizeof(b));
}
c=calloc(l,3);
for(i=0;i<l;i++)
{ *(c+i)=*4.0;}
for(i=0;i<l;i++)
{
    printf("%d %d %f %u %d\n",
        i, *(c+i), *(c+i), sizeof(c));
}
for(i=+1;i<l+1;i++)
{
    printf("%d %d %d %u\n",
        i, *(b+i), *(b+i));
}
}

```

So we want to, it is just a declaration right now. We want to allocate some memory to it, and then only we can put some value into it, so that is what we are trying to do here. So we say that, we have defined some variable l as 4, so that is and then this is the use of malloc function, so a is equal to integer star malloc of l okay. We will see this again in the and we will run this and see. Okay so here I have declared this. This is the same program.

(Refer Slide Time: 29:48)

```

/* Dynamic memory allocation */
#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++)
    {

```

10, 19 Top

So now we have a as, a is a pointer. Now that pointer is being allocated some memory, so if you have declared a as an array, for example, this is the substitute for the array, as we have said, instead of a static memory a dynamic memory. So let us

say, a, we want to put in 4 values. We could say that l is 4 here. So I could now declare the amount of memory which is to be allocated to that. So I know in this stage of the program, I know what, how much, what should be my array size is, okay so I can use that array size and allocate memory to this dynamically during execution. This is the way to do it. I said a is an integer, a is this array. This array is now an integer, integer type and it is in dimension l. So I am going to allocate memory for that okay. Then I can put in values to it. So I just now, this is the assignment statement, so I am putting in asterisk a plus i as i some number. Okay I just put it as i here. It could be any number. I am just assigning values to that array elements. Then I just print that out just to show you I print that out. I can print out the address and I can print out the values.

Again, 2 different ways of getting the array elements. Sorry, I print out the value of i which I put inside that and I put in the value of that element a plus I and the address of the element a plus i. So that is one way of assigning. Another way of assigning is now I have declared similar, exactly similar, another pointer, p, and I am showing you how you can allocate memory using calloc. So in calloc, in some compilers, the slight difference between calloc and malloc is that when you use calloc, you have to tell what type of variable that is inside this function, so but this is depends on the compilers, in modern compilers this is not required.

Another difference between malloc and calloc is that in malloc, when you allocate memory, and if you do not put in any variable, it puts in some garbage into it, while calloc initializes that to 0 if you do not put in anything. This is again compiler-specific okay. In most of the modern compilers there is hardly any difference between malloc and calloc. You could use both interchangeably. So here again, is exactly the same thing I have put p, I have allocated memory to p, and I put in p as instead of I, i star 2, and I can print out that. I can also use now this print statement. I am also using this function to demonstrate the size of p, that is, the number of elements the size of the entity, p.

(Refer Slide Time: 33:59)

```
Sample8.c
/* Dynamic memory allocation */
#include<math.h>
#include<stdio.h>
#define l 4
main()
{
int *a,i,*p;
float *b,*c;
a=(int*) malloc(l);
```


So we will run this and we can see this. So that is how I allocate it into a and p, the 2 integers and then I show you that I can allocate also, so that is for the integer type. Here I am allocating for a floating point type, So, as you can see here, that actually you can declare here what is the type, but it's not required in this case. So b equal to I, say malloc of i. I can do that and calloc would be, you can say, what the type is by putting the number here. If it is an integer, I would have put 2 and if it is a floating point number, I put 8. So that is 2 different ways of doing this malloc and calloc. Now this is a floating point. b and c are float variables and while p and a are integers. Okay so this is what we are going to demonstrate here and it is printing out these variables. So we will see that in this program.

(Refer Slide Time: 34:12)

```
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));
    }
```

(Refer Slide Time: 34:37)

```
b=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));
    }
```

Okay so now that is what we have printed out. Now we have to go through this one by one. So we have first used a. So then we have, as I said, we have put in 4 values into

it. So now that is the address, the values were just simply i, so a of 0 is 0, a of one was, you remember, a was put in as i itself. Okay so a of 0 is 0, a of 1 is 1, a of 2 is 2, a of 3 is 3 and the addresses, you can see the addresses printed out here. It goes by 4 okay, 36, 40, 44 and 48, and then we had the calloc function. We use here again, it is similar.

(Refer Slide Time: 35:00)

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

We again put in four values. We use the calloc as integer type, and we again print it out, and we set the size of that, the size of the array, we actually had 4, so the size of the array is 4. So we use the size of function. Remember, here we can see that when we are printing it out, in the case of p, we use the size of p. That is another print variable. So that has been given as 4 because we have allocated 4, l is 4 declared here, and we use calloc. We use l as 4, so we got 4 as size and the next part was assigning floating. Here, we had used b is equal to malloc of l, and b is now float. It is a floating point variable. Okay so then we again have the values. The values we put in here were i star 4, i 0 of 0 is 0, and then it jumps by 4. We have 0, 4, 8, 12, 16 and 20 okay.

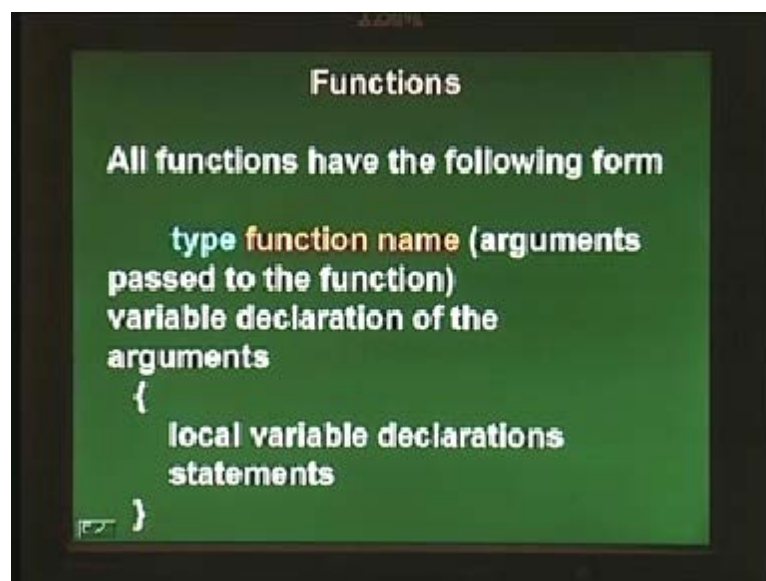
So now you notice that again the address changes by 4 and the size of function is again giving us 4, the dimension of that array and similarly, another thing I have declared using b, using calloc, the similar one and here we can see that again, but now I declare this thing and you get again the addresses, the variables and the size of functions. To use size of function, it does not matter whether you use malloc or calloc. It is just the size of the array. So we can use the size of function to pick up the size of the array which is being used. That is what this program demonstrates to us. Okay we should just look at the summary of the functions. We said we can pass array to a function. We can get an array from a function and we know how to declare arrays. We know how to declare pointer arrays. We know how to declare pointer arrays such that we can allocate memory dynamically and how to get the array size, things like that okay.

So Before we go into the next part of the pointers, that is, pointers to function, let us just go through the thing, let us review the idea of functions itself. So remember, all

functions have the following form. So next part we are going to do is actually looking at how to make a pointer. Now So far, we looked at pointer, at a one-dimensional array pointer, at a two-dimensional array and things like that and how to allocate memory. Now we want to know how to define a pointer to a function itself and how do we pass the pointer to a function to another function. Okay this will be required. We will see that where it will be required, and how we are going to use it.

So here, before we do that let us just summarize what we know about functions. All functions have the following form. The function, if it is returning something, we have to say what type it is, if it is not returning anything, we would say void and then we have the function name and the arguments passed to the function. This is a generic name in the function itself, that is the way we call it, declare it in the function itself. It is a similar declaration. We have the type the function, name, and the arguments passed to the function, and the arguments. The type of the arguments can be given inside these brackets or it could be given outside just below that. The variable declaration of the arguments can be included inside this bracket or we could have it outside. We saw that also. Okay so now then we have inside the function the local variable declaration and statements. That is what we saw. For example, print function, we saw statement which simply prints or something has to be returned then we say return these variables.

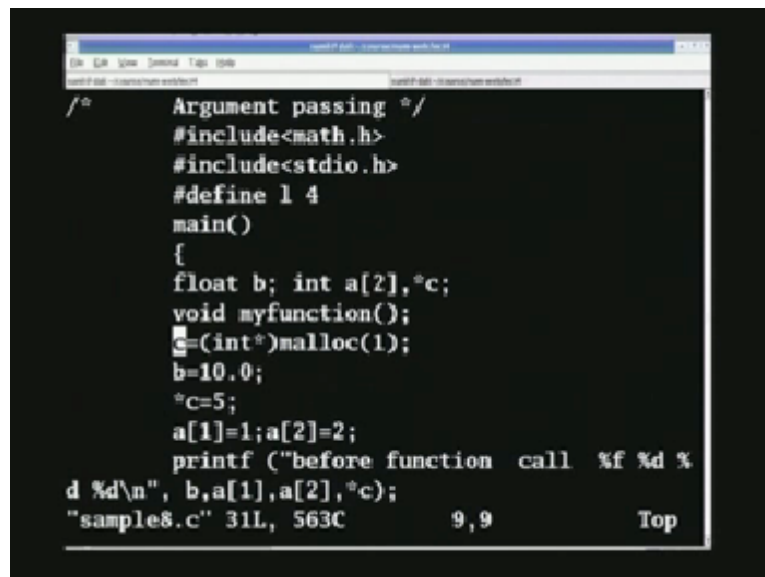
(Refer Slide Time: 39:47).



Okay so now so argument passing; what are the different ways of argument passing? We saw call by value, and said here the function, what will be the copy of the variable, and the changes made in the function will not affect the variable value in the function from which it is called. That is what we saw. It is called call by value, as supposed to call by reference, in which we pass the address to the variable and in this case, whatever changes which are made inside the function would affect, would change the value given in the function from which it is called, function of the main program from which it is called. So we saw that. So we will look at an example once again of this function calls. So here, again, this is the argument passing. We just saw

that. So here I am using also the malloc. So let us just say summary again. So we have here, we declared the function okay. This function is not going to return anything.

(Refer Slide Time: 40:50)



```
/* Argument passing */
#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[1]=1;a[2]=2;
printf ("before function call %f %d %d\n", b,a[1],a[2],*c);
}
"sample8.c" 31L, 563C          9,9          Top
```

So we declare a function called my function. So it is not going to return anything. So it is called void right. And then we have a variable floating point variable, we have an integer array, and we have a pointer, c. We have everything which we have learned about pointers here except two-dimensional array. Okay so now we are assigning some memory to that pointer, c. So we put some memory and then we have this floating point, another assignment statement. Floating point, b, is being given value 10, and this variable, c, is given a value 5. It is an integer. So it is declared as an integer. Here it is declared as an integer. And then we have the arrays, and we have given some values to the arrays too okay. What we do in the program is we print out all functions before these values, before we call a function, we are just printing out all the values. What are we printing out?

We are printing out the value of b percentage f because it is a floating point. We are printing out the value of a of 1. So that is percentage d. And then we are printing out a of 2. That is again an integer. So percentage d, and then we are printing out the value which is in the variable, c. We are putting some value in c, and that is again an integer, and then we call the function, and we print out exactly the same again. So now we are calling the function. We are calling the function with value b, b is just a floating point. So we are going to call this by value and we are passing an array. We know that an array is also a pointer to its base address, the array name actually. This is a call by reference. We have seen all arrays are called by reference, c is of course a pointer, so c is called by reference.

So now we will do a function. We call that function, and we print out the values of b, a and c after the function call. What we do in the function, inside the function, it is received as, so remember, we call this function as b, a and c and it is received as x, y and d. Now this is a function declaration. So this is void. It is not returning anything. It is function name variables, variable the declarations are it is a floating point, x is a

floating point, y is an array and d is a pointer. I could do exactly the same thing inside this bracket and avoid this line. These are 2 different ways of doing this. Then we have a local variable. Local variable is z, it is a floating point.

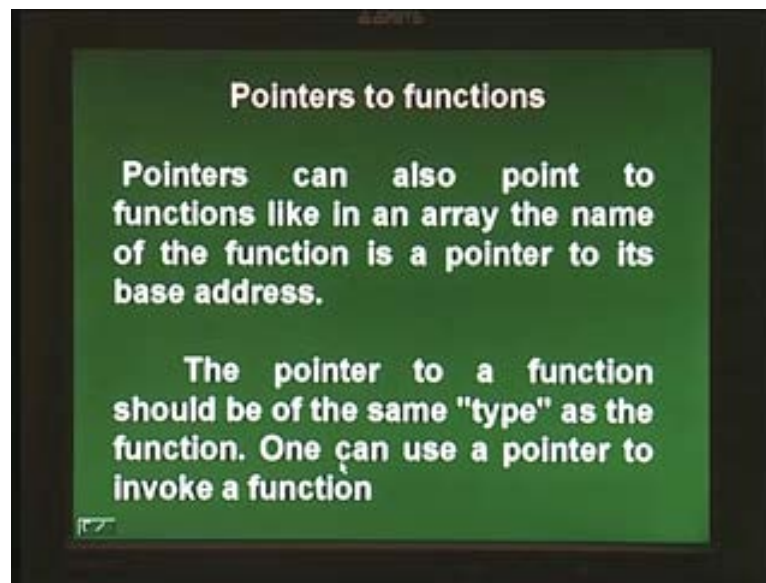
Now we are going to change x. x is passed by value. So x is b, is a floating point, and x is a floating point and this value, or b has been assigned to x and we change x here, and then we change y, we change y. This is an array, this is an array, we change the value of y. We change y to three times of y. Then we have d which is a pointer. We change the value of d. We were adding 2 to it. So that is what we do in the function, and then we are going to print out. This function is not doing anything else. It is not returning anything, it is not printing out anything. It is just changing these values, it changes the x, y and d values. So we are going to print out this before call and after call and then see what happens. Okay so before call, we had 10, 1, 2 and 5. Let us look at that again. We had given a 1 as 1 and a 2 as 2 and a c as 5 and b as 10. So that is what we had.

Okay so we had 10, 1 and 5. And what did we do in the function? In the function, we changed x which was b, b was passed as x. That is changed. We changed y and we changed d. That means, we basically changed, whatever is passed on to it is changed. Okay and then what do we see when we print out b, a and c after the function call? b is not changed because it was passed by value and we see that here a was passed as y and we changed y. But when we changed y, the a also changed. So we are going to change to 3 and 6. We multiplied y by 3, then a also changed because it was passed by reference because an array is a pointer to its own, to its base address. All arrays are passed by reference.

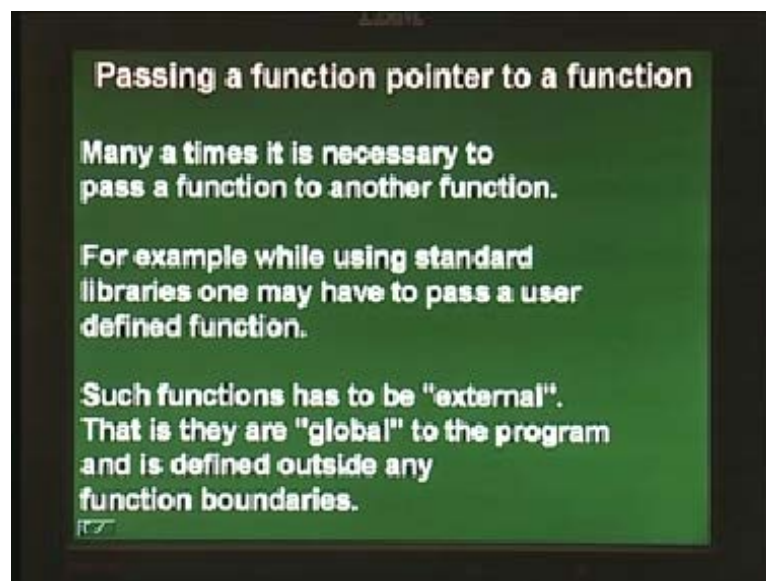
When you pass by reference, you change the value in the function. It will change the value in the main program. Then we see d is again an array, sorry a pointer. So it is again passed by reference. You change anything which is passed by reference inside a program, it changes in the main program. So we see that when we added 2 to d which is actually c passed as d, we added 2 to it. It changed the value of c, and that is what you can see here. This is not surprising because c and a are pointers to addresses, and what we are doing is we are just changing that value which is put in the address. So it just reads out what is the value there, and that has been changed, y, when you pass by value, that is like b which is like pass by value to x, x is a different memory location from b, and what you change in x is not going to change what is in that b. That is the point here.

So now we see the last part of the pointers. It is to see pointers to functions. This is also required many times, because we may want to pass a pointer, pass a function to another function. Okay so this is useful, and we can assign a pointer. We saw that pointers can be assigned to some variables. We equate the address. We can point use the pointer to point to the address of some variable, and we can also use, similar way, pointer to point to a function, like an array name, like in an array. Then we can use that pointer to invoke that function. Okay so the pointer to the function, the pointer to a function should be the same type as the function itself. This is important, similar as we used before.

(Refer Slide Time: 48:01)



(Refer Slide Time: 48:07)



When you are pointing a pointer to a particular memory location of some variable, they should be of the same type then we can use the pointer to invoke a function and then what is the use of this, one is that passing, so we can pass the function pointer to another function. This is especially useful when we are using some standard libraries and things like that, where some user defined function may have to be passed. In that case, we can use this property this function. We will see that. Okay how do we do this, defining a pointer to a function and how do we pass that function to another function? Only point to remember is that the function and the pointer should be of the same type. The function you are calling is of integer type, the pointer also has to be integer type. One more important point is that the function we are going to call by reference by referring it to a, by assigning it to a pointer should be declared outside the main program global to the whole program. We will see that in an example. That is the best way to see it again.

(Refer Slide Time: 49:16)

```
/*Pointer to a function and passing the
pointer to a function */
#include<math.h>
#include<stdio.h>
#define I 4
/*void myfunction (float x, int y[], int *d,
void(*cube)(float));
void cube(float);
main()
{
float b; int a[2], *e;
void (*func_ptr)();
void myfunction();
e=(int*)malloc(1);
b=10.0; *e=8; a[1]=1; a[2]=2;
func_ptr=myfunction;
printf ("before function call %f %d %d
%d\n", b,a[1],a[2],*e);
(*func_ptr)(b,a,e,cube);
printf ("after function call %f %d %d
%d\n", b,a[1],a[2],*e);
}

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

void cube(x)
float x;
{
float x;
x=x*x*x;
printf("%f\n", x);
}
```

Okay so here, for example, let us look at the main program. Let us forget about this and just look at the main program. So here I have some floating point declarations, and then there is an array, there is a pointer and there is a value b, etcetera. So here this is what I wanted to show you. So here is a function pointer. I call it a function pointer. Any name can be used, but it is a function pointer. So there is a bracket outside, so and this is a, the type is void, it is not returning anything. So if I want, remember my function program before, the same my function, so if I want my function pointer to point to my function, and then they have to be of the same type. That is the important point okay.

So that is, this is the program which we just saw. I am just using the same program which we just saw, and we are going to invoke that using the function pointer. Instead of that, the previous program I just called, if you just remember that I just called them by saying my function a, b, a, c. I will not do that here, I have declared a function pointer and I am going to point my function pointer to that, my function and I am going to invoke that function by using this, my function, by using the pointer name. That is another way of doing it okay. So that is what we would see here. I have added one more thing into that, this function. Here is that. Now I can also pass. Now there is another function which I have declared outside, which is called cube.

So this function has been declared outside. There are various ways of defining it. So this function will go into that. This function has been declared outside the main program, that is cube, and I can pass the address of that function, address to that function, into another function. That is my function. To my function, I am passing the address to that function. That is what we are going to do here, and then we will run this, and my function gets it. You see that my function is a program that gets it as whatever I call as cube here, which is defined outside the main program. It gets it as power 3. So let us say this is part of the standard library and it wants a particular name of the program there. It does not matter. You pass their address to that function. It goes into whatever name which is declared here okay. It is power 3 and inside this function I am calling power 3, again by pointer okay.

So this cube has passed as power 3 and power 3 is invoked there and then that is executed and printed. So that is what we are going to do. The function cube. So this is a user-defined function. Let us say this was a part of the standard package and it wanted a user-defined function to compute the cube and then what we do is we declare the function cube outside. We write the program, the user-defined program cube okay and we write that thing and then we put that thing back into, we put it in the main program. We call it. We declare it outside the main program and we call as cube, and this standard package gets it as power 3 and it computes that, and that is what we would see. So, we will just look at that.

So here is the same program. So I have declared the main function. This is the main function. What is important is, it does not return anything. And then we have the pointer to the function, and then my function itself, and when I call that function using this pointer, I pass the address to that user-defined function cube which is defined outside, which is written here. Now we will run this. We did 2 things here that is, we declared a function pointer and we also wanted to pass a pointer to a function, a function. So two things you wanted to do here. So that is what we did in this call. We used a function pointer here which is pointing to my function and we used that function pointer to invoke my function.

At the same time, inside this my function, I pass this, an address, to another function called cube. So that is what we are doing here. So we just run this now and we will see. So that is what we did. We now have one more thing here, that is like the earlier one. We had passed a variable by value, and then variable by reference and then we also passed the function. So we passed a name to a function. Okay so that function is called cube. Now that cube is printing out. As we said, the cube is printing out the cube of that. Okay so what is the function cube doing? It is not returning anything inside that function. It is just getting the value x, and the value x, the cube of that, is being printed out and that is what we see here. Okay so that is the cube of function.

So that is basically this program, do 2 things. To repeat it, that is, we use this function, this is the way to invoke a function using a pointer, so we can point something into a function and then use that pointer to invoke that function. And then we can declare some programs outside the main loop and then we can use the address to the main loop to the programs and pass that to another function. So we pass that to another function and then which would call it by reference using that pointer. So that is the summary of the pointer thing. So now I think the next lecture onwards we would probably put these things into use to do some actual numerical calculations.