

Problem Solving through Programming In C
Prof. Anupam Basu
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture – 59
Pointer (contd.)

So, we were discussing about pointers and we have seen that pointer is a variable; so, since it is a variable, but a variable that points to some other variable, but naturally the question that can arise is since pointers are variables.

So, you should be able to do some sort of operations like arithmetic operations on them. So, the answer is yes we can do that and in today's lecture we will look at exactly that pointer expressions.

(Refer Slide Time: 00:46)

Pointer Expressions

- Like other variables, pointer variables can be used in expressions.
- If p1 and p2 are two pointers, the following statements are valid:

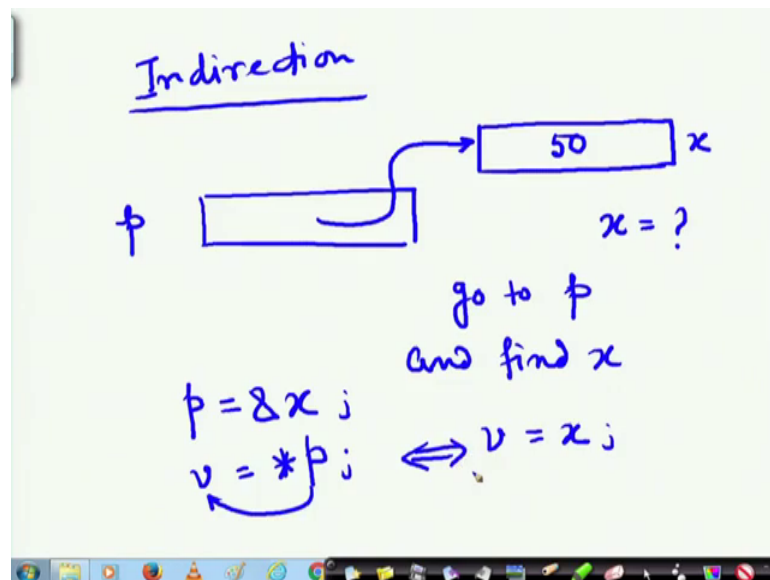
```
sum = *p1 + *p2;
```

The diagram shows two pointer variables, p1 and p2, each represented by a box with an arrow pointing to another box. p1 points to a box containing the value 50, and p2 points to a box containing the value 60. Below the code snippet, a blue arrow points from the value 110 to the variable sum, indicating the result of the expression *p1 + *p2.

Now, this concept of pointers is a very strong component of the c language and it is not the case that in all languages this pointer is there, but we are discussing pointers specifically, because it will give you a very good idea about what indirection is. We had mentioned about indirection right.

So, for example, I am just before moving into the actual discussion let me come to this that what is the indirection.

(Refer Slide Time: 01:20)



Somebody asks you the address of tom's house, you can do two things you can either give him the address of tom's house or you can give him the address of john's house. So, that he goes to john's house and asks john to get the address of tom's house. So, I do not know tom's address, but I know john who knows tom's address this is indirection one level of indirection.

Another level of indirection could be that second level of indirection, you can go to ram and ram will know the address of john who knows the address of tom. So, that is second level of indirection. So, in our case we have got a particular variable p let me just call this variable x , and that x has got some value. I am not saying an x has also got an address.

But I am not saying what is the value of x , what is x , that is my question. And instead of answering that, I am giving you a pointer to x and I am asking that I tell you I give you the answer go to p and find x right. So, then p must be assigned the address of x . So, I go to this, then I come to p and get the value of x as I want to have that in some other variable v , where I want to have star p .

So, indirectly I could have done simply v assigned x these two are equivalent, that is what we discussed in the last class right that is why it is an indirection that often comes in very handy very useful, when we carry out many computations.

So, like other variables pointer variables can also be used in expressions ok. If p 1 and p 2 are two pointers the following statements are valid star p 1 plus star p 2. So, what is being meant by that? Suppose p 1 p 1 is pointing to something, where there is 50 and p 2 is pointing to something which is stored as 60 then what is sum? Sum becomes 50 plus 60 star p 1 plus star p 2 50 plus 60 so, that should be 110 all right.

(Refer Slide Time: 04:55)

Pointer Expressions

- Like other variables, pointer variables can be used in expressions.
- If p1 and p2 are two pointers, the following statements are valid:

```

sum = *p1 + *p2;
prod = *p1 * *p2;
prod = (*p1) * (*p2);
*p1 = *p1 + 2;
    
```

$*p1 + 2$
 $\Rightarrow x + 2$
 $\Rightarrow 42$

Next similarly, I can have this, but here it is a little it will be nicer if I had it would be easier nicer to read, if I had put parentheses. So, that I had not confused about this asterix and this asterix these have got completely different significance, there is a multiplication and this is saying this asterix is saying it is a content of a particular pointer all right.

So, similarly the these two are equivalent of course, I have already shown that now this is also possible star p 1. So, in my earlier drawing p 1 was 50, 50 plus 2 that is being coming over here. So, it is this one looks like that, whenever you are finding difficulty as I suggested just draw a simple diagram p 1; p 1 is pointing to some other variable which has got the value 40.

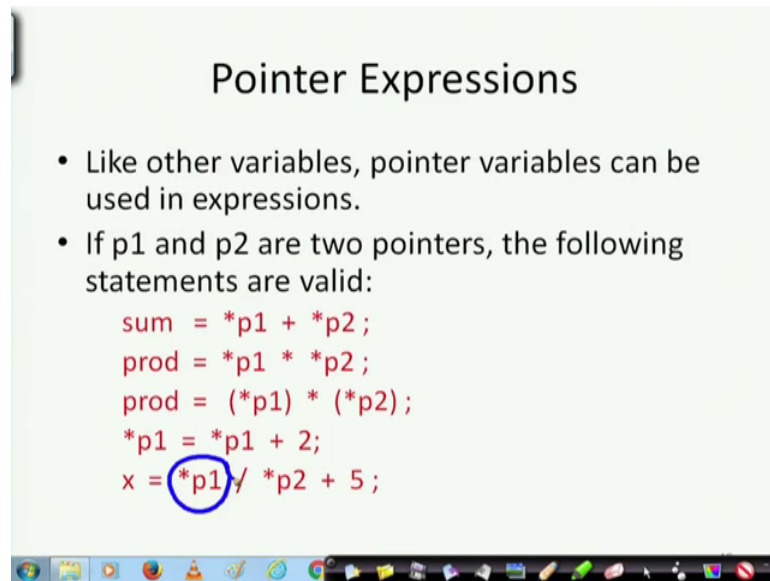
So, p 1 star p 1 is what? Star p 1 is that particular variable x and plus 2 that is equivalent to x plus 2 which is 42 all right and so, that is coming to. So, this one is being modified to 42 clear.

(Refer Slide Time: 06:29)

Pointer Expressions

- Like other variables, pointer variables can be used in expressions.
- If `p1` and `p2` are two pointers, the following statements are valid:

```
sum = *p1 + *p2;  
prod = *p1 * *p2;  
prod = (*p1) * (*p2);  
*p1 = *p1 + 2;  
x = (*p1) / *p2 + 5;
```


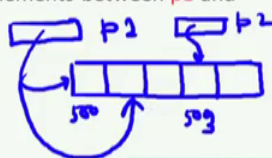


Similarly, I can do other operations like this where you must understand that this is actually just like another variable that an integer variable or whatever type `p1` is, that type of variable and it is simple no other complications in that.

(Refer Slide Time: 06:48)

Contd.

- What are allowed in C?
 - Add an integer to a pointer.
 - Subtract an integer from a pointer.
 - Subtract one pointer from another (related).
 - If `p1` and `p2` are both pointers to the same array, then `p2-p1` gives the number of elements between `p1` and `p2`.
- What are not allowed?

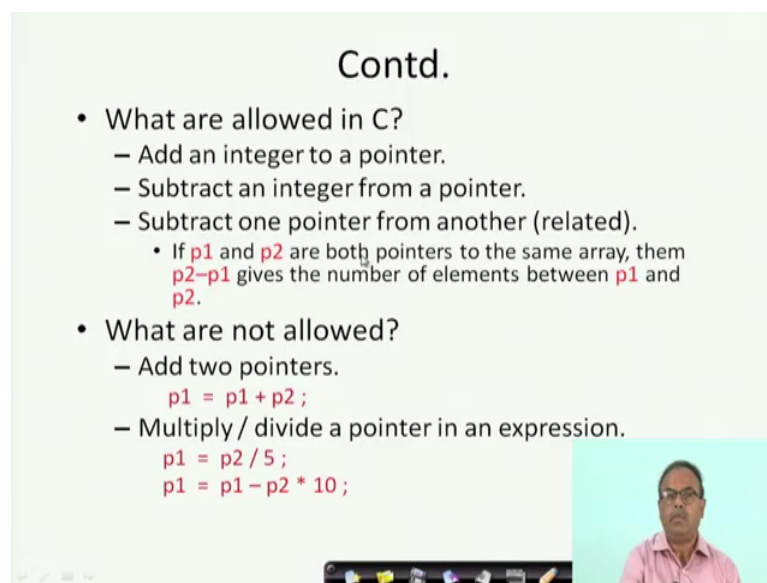


Now, what are allowed in C? The certain things are allowed in C and some things are not allowed in C, I can add an integer to a pointer, I can subtract an integer from a pointer, I can subtract one pointer from another and say if `p1` and `p2` are both pointers to the same array, then `p2` minus `p1` gives the number of elements between `p1` and `p2`. For

example, suppose here there is an array. So, p 1 is pointing here and p 2 is pointing here then the number of elements p 2 minus p 1 will be just the subtraction of these addresses.

Suppose this was 5000 500 501 if it be a character 501, 502, 503. So, I have got three elements in between right. So, these are all allowed I can subtract an integer from a pointer I can add an int add an integer to a pointer. So, if I add an integer to a pointer that is p 1, p 1 plus 1. So, that means it will point to this point, this element right what are not allowed? The things that are not allowed are you cannot add two pointers.

(Refer Slide Time: 08:22)



Contd.

- What are allowed in C?
 - Add an integer to a pointer.
 - Subtract an integer from a pointer.
 - Subtract one pointer from another (related).
 - If p1 and p2 are both pointers to the same array, then p2-p1 gives the number of elements between p1 and p2.
- What are not allowed?
 - Add two pointers.
p1 = p1 + p2 ;
 - Multiply / divide a pointer in an expression.
p1 = p2 / 5 ;
p1 = p1 - p2 * 10 ;

The reason is obvious p 1 is a pointer and p 2 is a pointer now these two are two different locations. So, say p 1 is in location 7000 pointing to some variable, and p 2 is in location 10,000 pointing to some other variable.

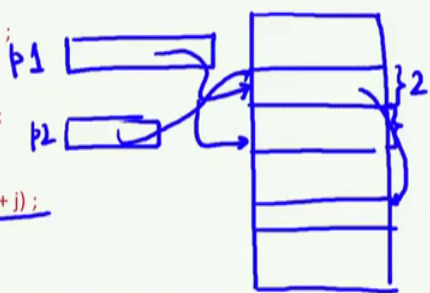
So, what does p 1 plus p 2 mean? 7,000 plus 10,000 17,000 that does not mean anything that can be a point to some garbage value or something else; So, that is not allowed compiler we hold you for that, multiply or divide a pointer in an expression that is also not allowed. You cannot multiply you can just add an integer, subtract an integer or subtract one pointer from another these three you can do.

(Refer Slide Time: 09:14)

Scale Factor

- We have seen that an integer value can be added to or subtracted from a pointer variable.

```
int *p1, *p2;  
int i, j;  
:  
:  
p1 = p1 + 1;  
p2 = p1 + j;  
p2++;  
p2 = p2 - (i + j);
```



And here is something which is known as a scale factor, let us see whether you understand this or not we have seen that an integer value can be added or subtracted.

So, here let us look at this, p 1 and p 2 are two pointers of type integer. I mean when p 1 is pointing to an integer p 2 is also pointing to an integer all right. Now, idea to integer variables p 1 is p 1 plus 1 so; that means, what p 1 was pointing somewhere, but I am just adding some constant to that. So, here is a memory location and say p 1 is pointing to this all right and suppose it is an integer. So, this is pointing to an integer. Now p 1 plus one means this will point to the next integer.

Now, I am not saying whether if an integer takes two bytes each of them are of two bytes. So, it just comes to the next integers, the pointer arithmetic case that is why the type is important. Depending on the type it is updating either by 2 or by 1, but p 1 plus 1 means I am going to the next. I am p 2 plus p p 2 assigned p 1 plus j where j can be something.

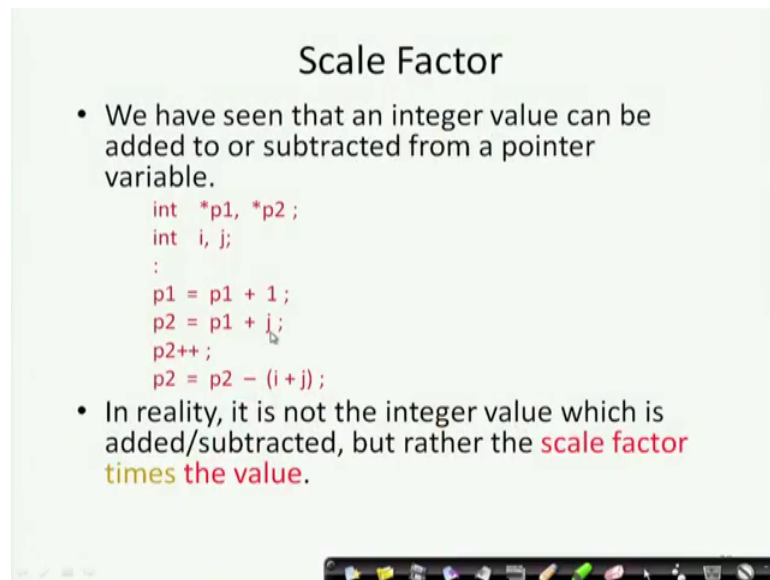
So, it goes to the. So, p 2 was pointing somewhere here, I am upgrading that with the j value going to the j th next similarly p 2 plus plus p 2 assigned any arithmetic operation I can do all right.

(Refer Slide Time: 11:15)

Scale Factor

- We have seen that an integer value can be added to or subtracted from a pointer variable.

```
int *p1, *p2;
int i, j;
:
p1 = p1 + 1;
p2 = p1 + j;
p2++;
p2 = p2 - (i + j);
```
- In reality, it is not the integer value which is added/subtracted, but rather the **scale factor times the value**.




Next in reality it is not the integer value which is added or subtracted, but rather the scale factor times the value that is one means one times the size of the integer if it were. So, j times the size of an integer, 2 bytes, 4 bytes that is what that is why it is called the scale factor. So, this is not this 1, but next one next, two next, here j th next like that.

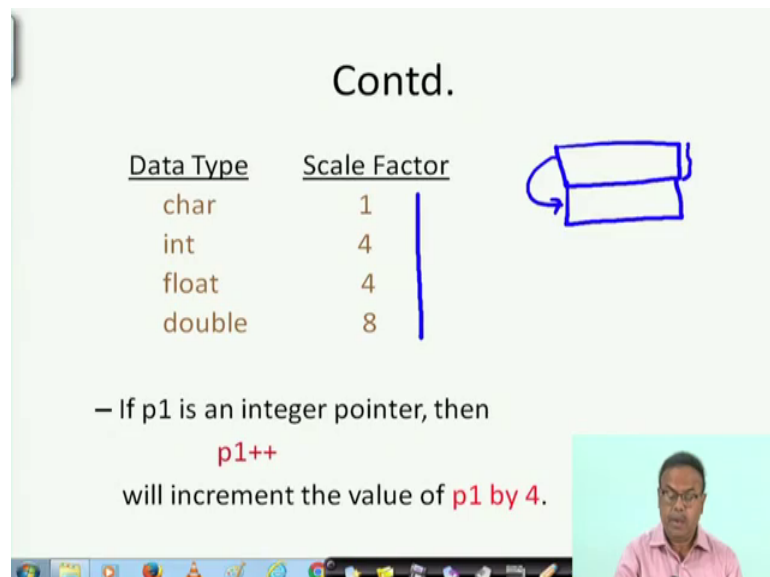
(Refer Slide Time: 11:50)

Contd.

<u>Data Type</u>	<u>Scale Factor</u>
char	1
int	4
float	4
double	8



– If p1 is an integer pointer, then
p1++
will increment the value of **p1** by 4.



So, for character the scale factor is 1, integer is 4, if 4 bytes take one integer float 4 now this could be 2 that depends on what the scale factor is depends on the particular machine. So, So, if I write here for an integer pointer, assuming the my computer is

actually doing this that each of them is 4 then p 1 plus plus is adding p 1 by 4 that is going to the next integer going to the next integer.

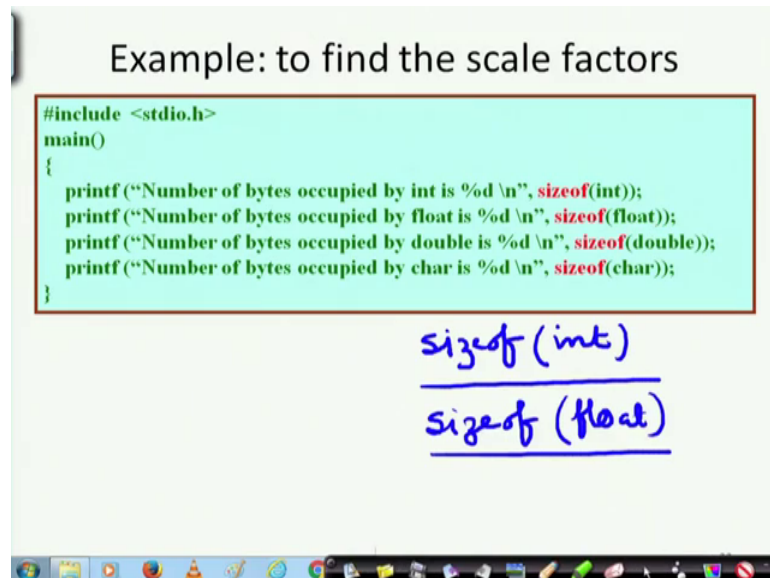
Now, in between there are 4 bytes ok. So, I am going to the next integer.

(Refer Slide Time: 12:31)

Example: to find the scale factors

```
#include <stdio.h>
main()
{
    printf("Number of bytes occupied by int is %d \n", sizeof(int));
    printf("Number of bytes occupied by float is %d \n", sizeof(float));
    printf("Number of bytes occupied by double is %d \n", sizeof(double));
    printf("Number of bytes occupied by char is %d \n", sizeof(char));
}
```

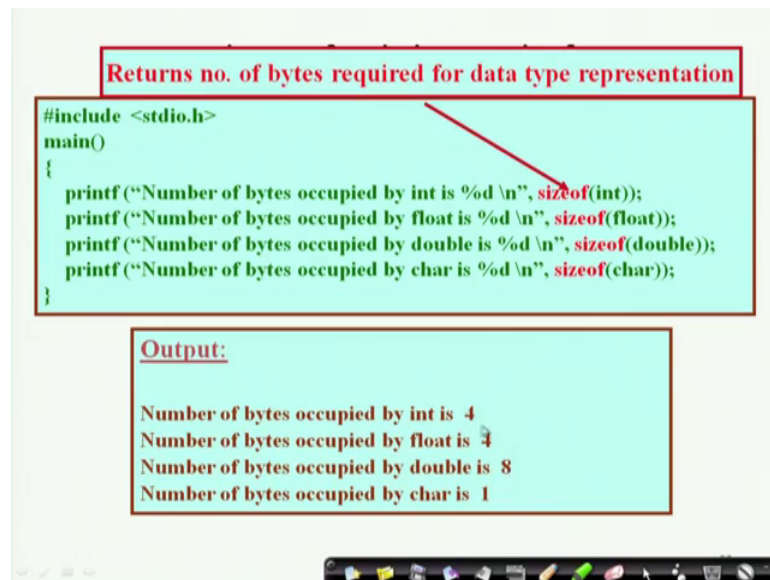
sizeof(int)
sizeof(float)

The image shows a presentation slide with a light green background. At the top, the title "Example: to find the scale factors" is written in black. Below the title is a code block with a light blue background and a red border, containing C code that uses the sizeof operator to print the byte sizes of int, float, double, and char. Below the code block, the expressions "sizeof(int)" and "sizeof(float)" are handwritten in blue ink, each underlined with a blue line. At the bottom of the slide, a Windows taskbar is visible with various application icons.

So, there is one quick way of finding out, how we can find out what is my what is the representation in my system. There is a nice inbuilt function called size of. So, if I give size of int, the system returns me the value 4 or 2 depending on how much, how many bytes does int consume.

Similarly, I could have given size of float that will tell me how many bytes is a float consumed so and so forth. So, that is one way to find the scale factors. So, number of bytes occupied by float is size of float, if you give that the system will give you the size of returns that one that value the number of bytes required for that representation.

(Refer Slide Time: 13:28)



Returns no. of bytes required for data type representation

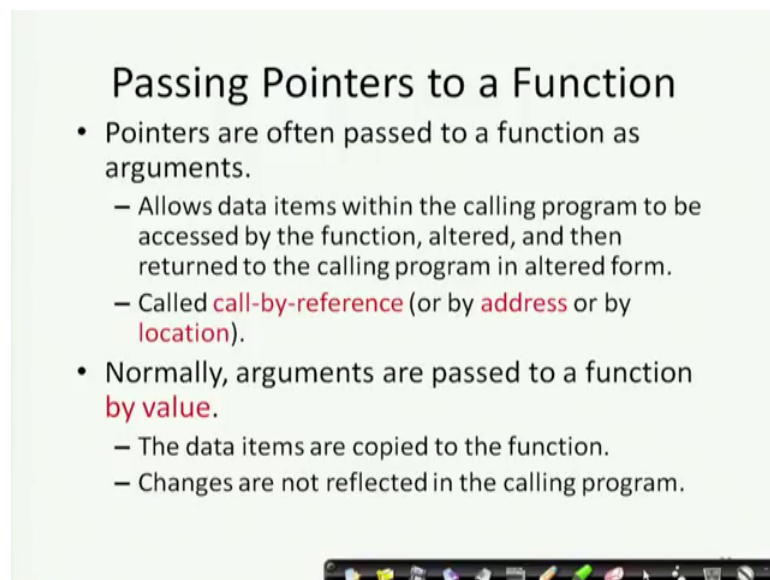
```
#include <stdio.h>
main()
{
    printf("Number of bytes occupied by int is %d \n", sizeof(int));
    printf("Number of bytes occupied by float is %d \n", sizeof(float));
    printf("Number of bytes occupied by double is %d \n", sizeof(double));
    printf("Number of bytes occupied by char is %d \n", sizeof(char));
}
```

Output:

```
Number of bytes occupied by int is 4
Number of bytes occupied by float is 4
Number of bytes occupied by double is 8
Number of bytes occupied by char is 1
```

So, if in a system you would run this and you find that it is float is 4 int is 4 then you know what my scale factor is.

(Refer Slide Time: 13:40)



Passing Pointers to a Function

- Pointers are often passed to a function as arguments.
 - Allows data items within the calling program to be accessed by the function, altered, and then returned to the calling program in altered form.
 - Called **call-by-reference** (or by **address** or by **location**).
- Normally, arguments are passed to a function **by value**.
 - The data items are copied to the function.
 - Changes are not reflected in the calling program.

Now, just like and for every case we are thinking of how do we pass an array to a function, how do we pass a structure to a function, here again we look at how do we pass a pointer to a function. Pointers are often passed as parameters to a function and if you have thought about it you must have already discovered. Now, always it allows the data

items, within the calling programs to be accessed by the function altered and then returned to the calling function in the altered form, this says the calling by reference ok.

Normally arguments are passed to a function by value, we have discussed we had discussed this. Now this is called as call by reference or call by address, now you can see this how this is done. Because in call by value we have seen that in the swap function for example, it was swapped within the function, but that x y and the main functions x y were two different entities therefore, whatever change was there that was lost.

So, therefore, here, but if I had just passed on the pointer then whatever change I do in the pointer in that particular location, I simply pass on the address and make a change over there then the change is reflected it is because it is the same location that is known as call by reference.

(Refer Slide Time: 15:15)

Example: passing arguments by value

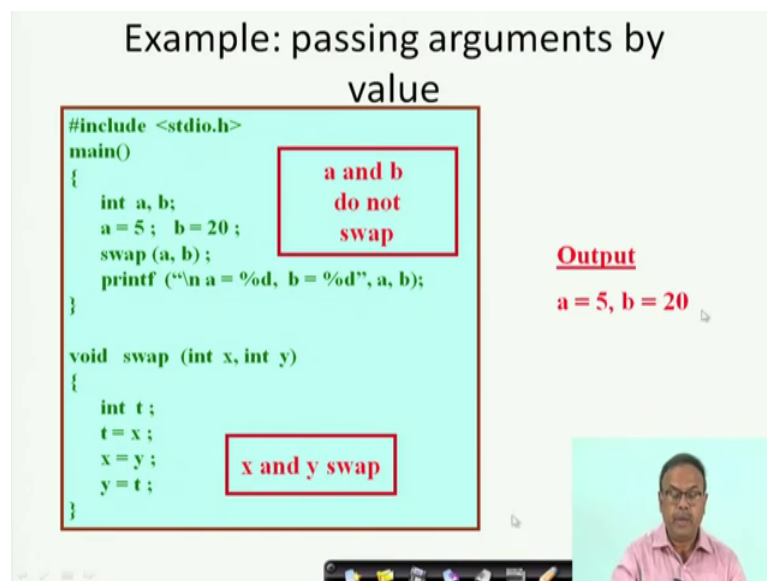
```
#include <stdio.h>
main()
{
    int a, b;
    a = 5 ; b = 20 ;
    swap (a, b);
    printf ("\n a = %d, b = %d", a, b);
}

void swap (int x, int y)
{
    int t ;
    t = x ;
    x = y ;
    y = t ;
}
```

a and b do not swap

x and y swap

Output
a = 5, b = 20



Now, here is an example of you have seen this passing by value, I am repeating this a was 5 and b was 20, I call swap a b here what happened it took x was 5 b was 20 I changed it. Now x was therefore, now x was 20 and b was 5, but when I returned and printed this a and b they were completely different. So, swapping was not reflected therefore, here x and y swap, a and b do not swap, but instead. So, the output will be the same 5 and 20.

(Refer Slide Time: 15:57)

Example: passing arguments by reference

```
#include <stdio.h>
main()
{
    int a, b;
    a = 5 ; b = 20 ;
    swap (&a, &b);
    printf ("\n a = %d, b = %d", a, b);
}

void swap (int *x, int *y)
{
    int t;
    t = *x ;
    *x = *y ;
    *y = t;
}
```

The diagram shows two memory locations, 'a' and 'b'. Location 'a' contains the value 5 and the address 20. Location 'b' contains the value 20 and the address 5. Below these are two empty memory locations labeled 'x' and 'y'. Arrows indicate that 'x' points to the address 20 (the address of 'a') and 'y' points to the address 5 (the address of 'b'). A handwritten note 't=5' is written above the diagram.

But if I had done through reference, here you see how I pass it on here what I am passing you have must have thought about it. Swap instead of sending that value, what I am sending is the address of that address of a and address of b and here inside the function, what I am a what am I accepting? I am accepting the pointer.

So, here x and y I know that what is coming to me is a pointer. So, what happened here a was 5 we discussed it earlier also, but let me repeat because there is a very fundamental idea, b is 20 and now I swap. So, here I have got do I have x? No I have got this x which is nothing, but and a and I have got this y which is nothing, but and b. So, therefore, they are pointing to these points.

Now, when I swap I am actually swapping the content of x ; that means, here that is going to t and then content of y content of y in direction, I go from here follow my cursor I go here and that one is going as the content of a content of x. So, here it is becoming 20 and then t is coming as a content of y what is y y is here. So, this is coming as 5.

So, when I come out and print here a and b as you can see has changed. So, you see how did I pass on the parameter? Look at this I have passed on the address and I have accepted them in my function as the pointer.

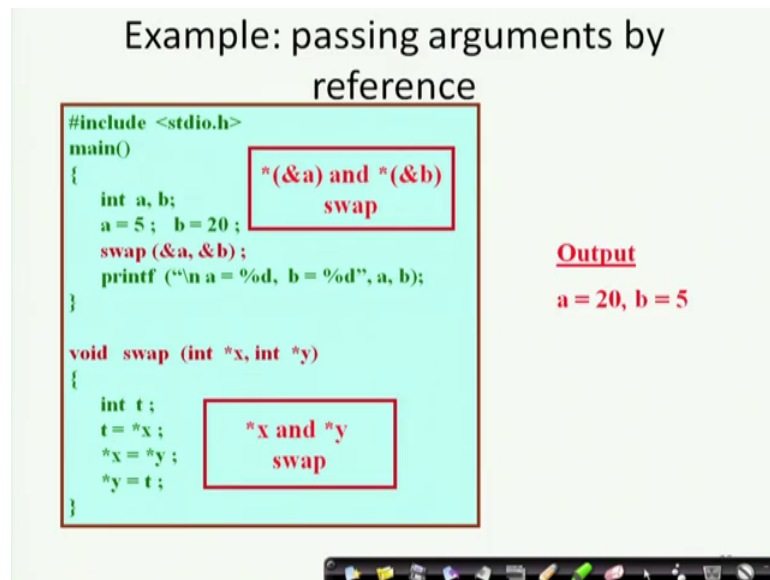
(Refer Slide Time: 18:12)

Example: passing arguments by reference

```
#include <stdio.h>
main()
{
    int a, b;
    a = 5 ; b = 20 ;
    swap (&a, &b);
    printf ("\n a = %d, b = %d", a, b);
}

void swap (int *x, int *y)
{
    int t ;
    t = *x ;
    *x = *y ;
    *y = t ;
}
```

Output
a = 20, b = 5

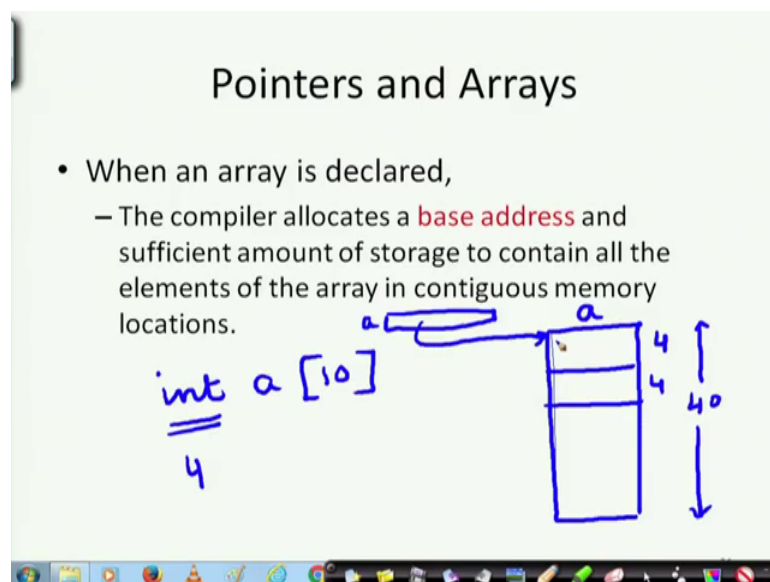


So, with this we will get the correct answer x and y swap a and b also swap. So, the answer is that is correct one as we expected. So, now let us skip this a little bit and let us go to something else as pointers and arrays.

(Refer Slide Time: 18:36)

Pointers and Arrays

- When an array is declared,
 - The compiler allocates a **base address** and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.

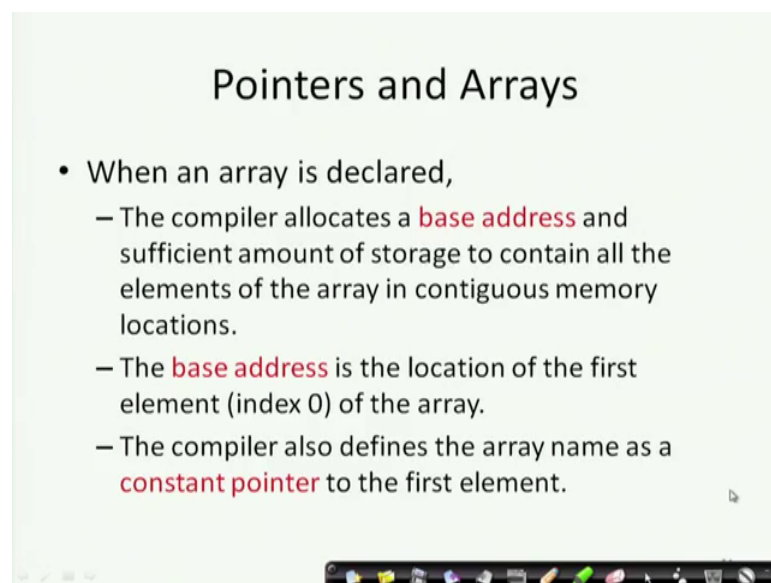


When an array is declared, when an array is declared the compiler allocates a base address and sufficient amount of storage. You know that when I declare something an array, int a 10. And if I say that int I have discovered using size of that int takes 4 bytes,

then for every element 4 bytes are kept and 40 such locations are allocated for me for me means for the array a.

Now, this is all right now when I refer to this array a, because we saw that we pass on an array by reference to a function when I pass an array, we call it by reference the reason behind that is. That this name array a is the same as a pointer to the first location of the array. So, it is as if equivalent to a is a pointer that is pointing to the first location of this array a all right they are equivalent.

(Refer Slide Time: 20:04)




So, the base address is a location of the first element of the array, the compiler also defines the array, name as a constant pointer to the first in constant pointer when I declared it with the compiler also keeps a constant pointer that pointer you cannot change ok.

(Refer Slide Time: 20:25)

Example

- Consider the declaration:
`int x[5] = {1, 2, 3, 4, 5};`
 - Suppose that the base address of x is 2500, and each integer requires 4 bytes.

<u>Element</u>	<u>Value</u>	<u>Address</u>
x[0]	1	2500
x[1]	2	2504
x[2]	3	2508
x[3]	4	2512
x[4]	5	2516



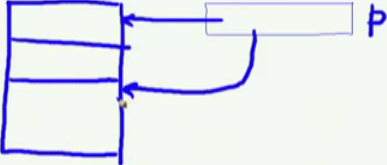
For example, when I say x 5, 1, 2, 3, 4, 5 then suppose the base address is 2500 each integer requires 4 bytes then the elements will be x 0 will be 2500, x 1 will be 2504 so and so forth and the pointer will be 2500.

(Refer Slide Time: 20:43)

Contd.

`x ⇔ &x[0] ⇔ 2500;`

- `p = x;` and `p = &x[0];` are equivalent.
- We can access successive values of x by using `p++` or `p--` to move from one element to another.

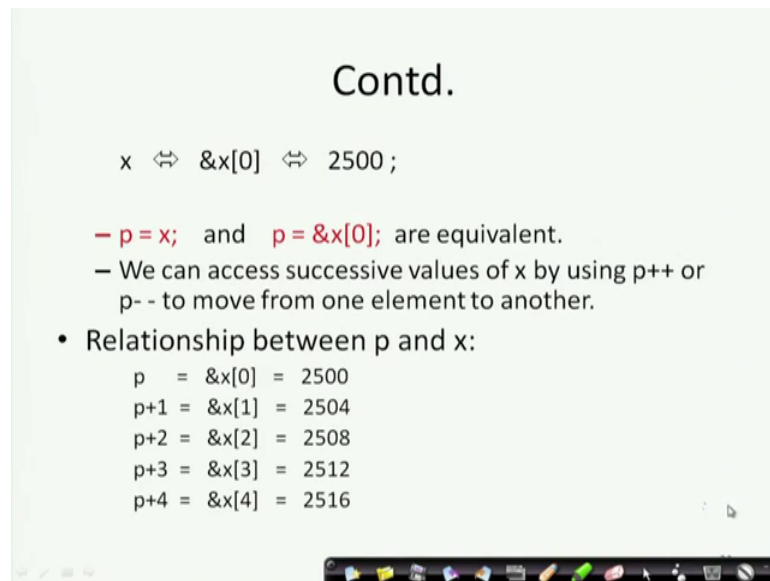


If we go up 2500 so, there is a pointer x means and address of x 0, which is 2500 all right.

So, p assigned x and p assigned end of x 0 are equivalent. So, we can access successive values of x by using p plus plus or p minus minus to move from one element to another.

So, I have got the p, I have got this pointer is pointing to the array. I cannot change that pointer, but if I do p plus plus this pointing to the first element, then I go to the next element of the array record by the scale factor ok. So, this is if I do p plus plus I am actually doing p plus 4 say in that way I can move across.

(Refer Slide Time: 21:56)



Contd.

`x ⇔ &x[0] ⇔ 2500 ;`

- `p = x;` and `p = &x[0];` are equivalent.
- We can access successive values of x by using `p++` or `p--` to move from one element to another.
- Relationship between p and x:
 - `p = &x[0] = 2500`
 - `p+1 = &x[1] = 2504`
 - `p+2 = &x[2] = 2508`
 - `p+3 = &x[3] = 2512`
 - `p+4 = &x[4] = 2516`

So, the relationships should be clear here. So, p plus 1 is a next one p plus 2 is the next one that we have already explained.

(Refer Slide Time: 22:02)

Example: function to find average

```
#include <stdio.h>
main()
{
    int x[100], k, n ;

    scanf ("%d", &n) ;

    for (k=0; k<n; k++)
        scanf ("%d", &x[k]) ;

    printf ("\nAverage is %f",
           avg (x, n));
}
```

```
float avg (int array[ ],int size)
{
    int *p, i, sum = 0;
    p = array ;
    for (i=0; i<size; i++)
        sum = sum + *(p+i);
    return ((float)sum / size);
}
```

So, here is a function to find average, here you see we have got an a main program, where I have got an array 100 elements for k assigned k to n for k 0 to n I am reading k and then I am calling this average x. I am calling average x n. Now, here what goes average x n you have seen this now I am passing the pointer. When I am saying I am just passing the array actually I have passed on the pointer. So, whatever I do I am doing here I am taking another star p, which is local which is pointing to the array.

So, my array was here and I am putting another pointer p which is pointing to this array and array means what? Array means the first location of the array then I carry on the sum here I carry on with p I here you see what I do I take p and then I change p some assigns star p plus i. So, I p plus 1, p plus 2, p plus3 and star p plus 1, p plus 2 means the content of this these contents.

So, here in this way I am getting the sum and returning what do I return? Return float sum by size. So, I get sum; obviously, the array was integer, but now this is something called typecasting that I can make it although it was float, I put it in a bracket. That means, whatever is coming here I am converting that to float all right sum will be the array sum of all integers will be an integer, but when I divide by that although I did not declare sum to be ah a floating point number, just by this sort of type typecasting.

I can make it a I will first do this and convert it, make it a floating point number and then how do I return it? Yes you might have guessed correctly, that I do not need to return it

because whatever I have done here when I passed an array, whatever was done that is being done here and this sum what I am returning? I am returning this to the average. So, I will get this value all right. So, this should be clear.

(Refer Slide Time: 25:15)

Example: function to find average

```
#include <stdio.h>
main()
{
    int x[100], k, n ;

    scanf ("%d", &n) ;

    for (k=0; k<n; k++)
        scanf ("%d", &x[k]) ;

    printf ("\nAverage is %f",
            avg (x, n));
}
```

```
float avg (int array[ ],int size)
{
    int *p, i , sum = 0;

    p = array ;

    for (i=0; i<size; i++)
        sum = sum + *(p+i);

    return ((float)sum / size);
}
```

*int *array*

p[i]

So, clarified now next thing that I will be discussing a little bit is on dynamic memory allocation, I will take little time to explain that that is a very important concept and after that we will move to a some discussions basic discussions on file.