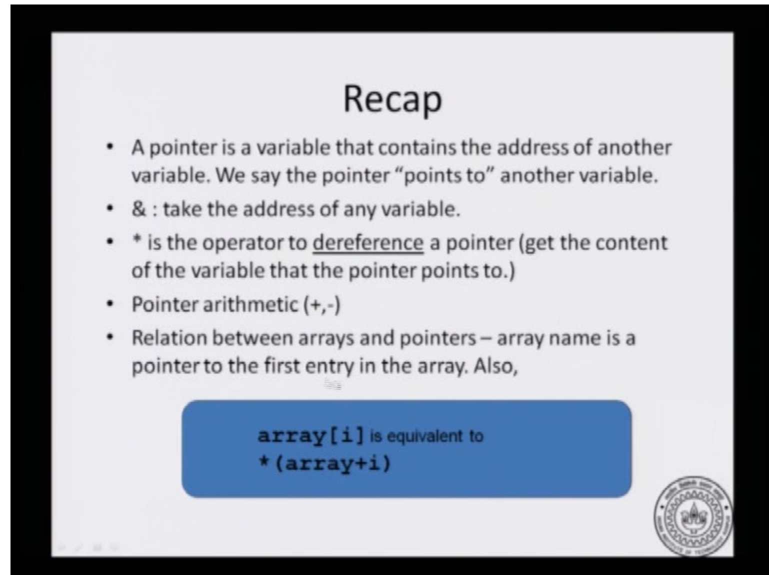
Introduction to Programming in C
Department of Computer Science and Engineering

(Refer Slide Time: 00:05)



So, here is the stuff that we have seen about pointers. First we have defined what is a pointer? A pointer is just a variable that holds the & another variable. We say that pointer points to another variable. And depending on what variable it points to, the type of that target, we say it is an int pointer or a character pointer or a float pointer and so on. So, this is the first thing what is a pointer? And then we have seen what all can you do with a pointer; what are the operations that you can do in a pointer. So, if you have a normal variable, you can take the &that variable using the & operator. If you have a pointer, then you can dereference the pointer by using *(ptr). That will go to the location pointer 2 by ptr and take the value of that target.

Further we have seen pointer arithmetic involving + and -. And I have introduced you with the caution that they are meant to navigate within arrays; they are not meant to navigate to arbitrary locations in the memory. If you do that, it may or may not work. And further we have touched up on the intimate relationship between arrays and pointers in C. As captured by the formula, array[i] is *(array+i). A special case of this is to say that the name of the array is an &the first entry in the array. For example, array[0] is the same as *(array+0). We have seen this and think about them once more to get comfortable with the notion.

(Refer Slide Time: 01:47)

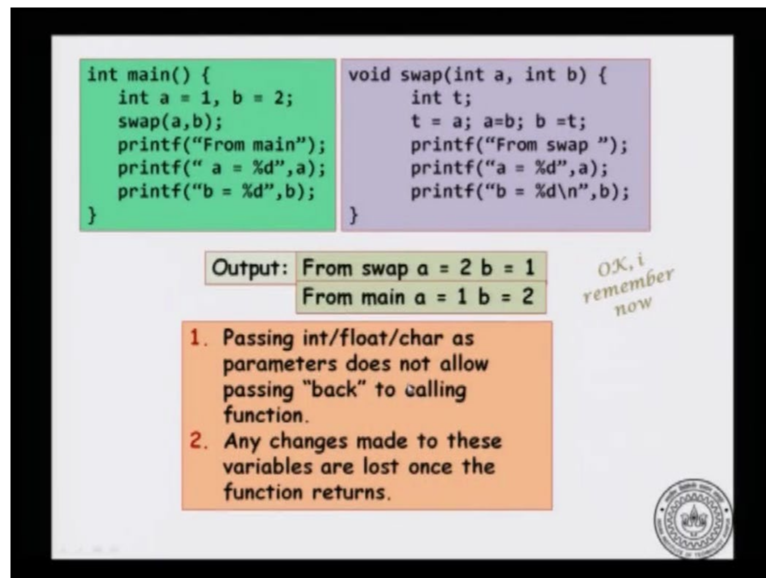
```
int main() {
    int a = 1, b = 2;
    swap(a,b);
    printf("From main");
    printf(" a = %d",a);
    printf("b = %d",b);
}
```

```
void swap(int a, int b) {
        int t;
        t = a; a=b; b =t;
        printf("From swap ");
        printf("a = %d",a);
        printf("b = %d\n",b);
}
```

In this video, we will talk about how pointers interact with functions. When we introduced arrays, we first said here are arrays; here is how you write programs with arrays. And then we introduced… Here is how you pass arrays into functions. Let us do that the same thing with pointers. So, here are pointers. And how do you pass them to pointers? Before coming into how do you pass them to pointers, we will go into – why should you pass pointers to functions. So, let me introduce this with a very standard example. This is a classic example in C. How do you exchange two variables? We have seen the three-way exchange; where, I said that, if you have three rooms: A, B… I have two full rooms: A and B.

And then I want to exchange the contents of these rooms; then I can use a third room. First, move the contents of A to T; that is your first move. Then move the contents of B to A; that is your second move. And then afterwards, A now contains the contents of B; and B is empty; T is containing the contents of A. So, the third move is – move T to B. So, the net effect will be that, B contains the whole contents of A; A contains the whole contents of B; B contains the whole contents of A; and T is empty. So, this was the three-way exchange, which we did within main function. This is long back when we discussed GCD algorithm.

(Refer Slide Time: 03:44)-

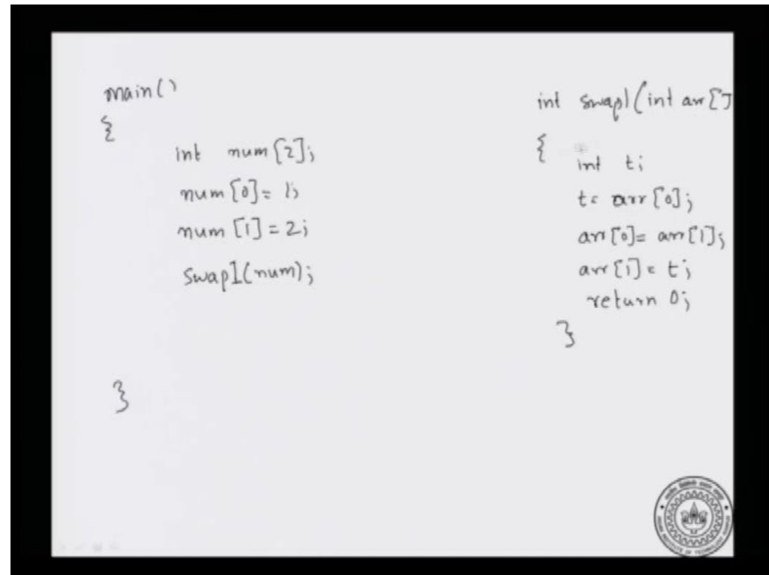


Now, let us try to do that using a function. So, I have a swap routine, which takes two integer arguments: a and b; and it is meant to exchange the values of a and b. So, inside main, I have a = 1, b = 2. And I call swap a and b. And swap a and b – what it does is this three-way exchange that, we have discussed. Now, just to test whether things are working, I have a bunch of printf statements, which says what is the value of swap, what is the value of a and b after swap has executed. Similarly, when I come back, I will just print the values of a and b to see what has happened after swap. So, when you call swap and you output it within swap, it is very clear that, a = 2, b = 1. So, the three-way exchange would work as you expect. And you have whatever was passed, which is swap1, 2. So, it will exchange those variables and it will print a = 2 and b = 1.

Now, within main, a was 1 and b is 2. Now, when you print these statements inside main, surprisingly, you will find that, a = 1 and b = 2. So, the effect of swap is completely absent when you come back to main. Within swap, they were exchanged. But, when you come back to main, they were not exchanged. Why does this happen? This is because remember that, some space is allocated to a function; and whatever space is allocated to the swap function, all the variables there is erased – are erased once you return from the swap function. So, within swap function, a and b are exchanged. But, all that is gone when you return to main. So, passing integer, float, character variables as parameters, does not allow passing back to the calling function; you have only the return value to return back. Any changes made within the called function are lost once this function

returns. So, the question is can we now make a new function such that work done within that function will be reflected back in main.

(Refer Slide Time: 06:39)

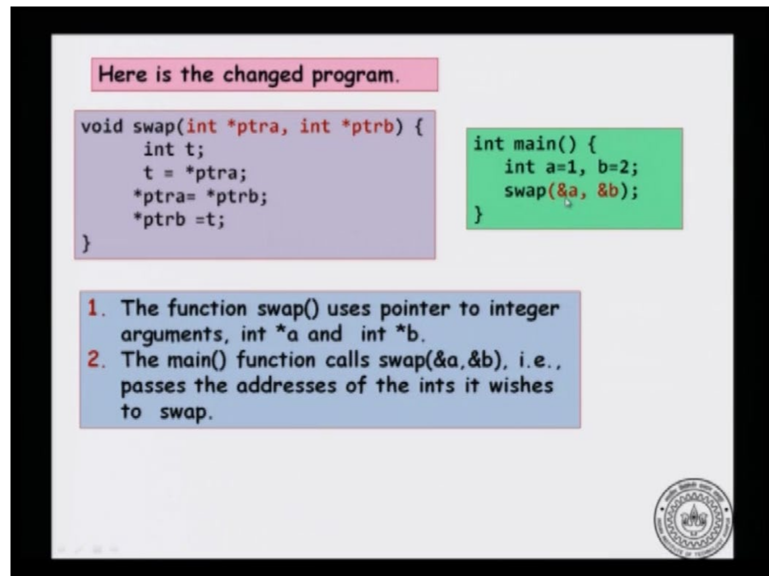


Now, here is an intermediate solution. We know that, if we pass arrays, then work done in the called function will be reflected back in the calling function. So, you could think of the following intermediate function. So, if I have int num 2 and then I say that, num[0] is 1, num[1] is 2. This is in the main function. And then I call swap of num. Now, we will call it swap1(num); I have a new function. Now, what swap1 does is – so int swap1 int arr. So, suppose I have this function; inside that, I will just say that, I will have an intermediate variable t; and then have t = num or arr[0]. Then arr[0] = t; arr[0] = arr[1]; and arr[1] = t. Suppose I have this function. And now, you can sort of argue that, this will also swap the two cells in the num array. So, the dirty trick that I am doing is that, I want to swap two variables; instead, I will say that, instead of these two variables, I will insert them into a array of size 2; and then call swap1 on that array. Now, what swap1 does is – it will exchange – it will do the three-way exchange on the array.

Now, I know that because of the way arrays are passed in C, any change that happens to the array arr inside swap1 will be reflected back in main. So, when I print these num array back in main, I would see that, num[0] is now 2 and num[1] is 1. So, this is an intermediate trick in order to write the correct swap function. But, you will agree that, this is a kind of a dirty trick, because in ordered to swap two variables, I created an array;

and then depended on the fact that, swap will change the array in such a way that, the change will reflected back in main. So, is there a nicer way to do it? That is what we are interested in. And the answer is let us just think about that array trick. What we did was – when we passed an array, we were of passing the &the array.

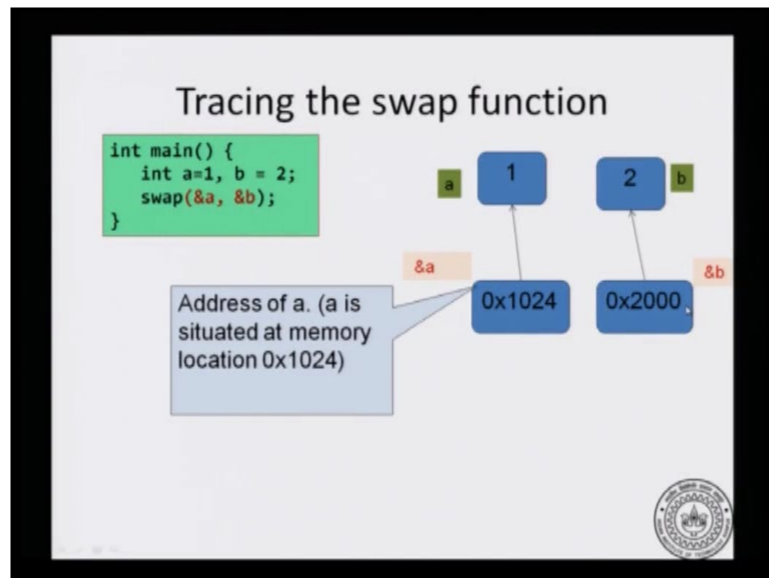
(Refer Slide Time: 09:40)



This is how arrays are passed to functions. So, now, let us just take that idea that, we are passing the address. So, let us try to write a swap function, where you are passing the &variables instead of the variables themselves. So, here is the correct swap function. And what I write is void swap. So, void is a new keyword that you will see; but it is not a big deal; it is just a function that does not return a value; it just performs an action without returning a value. So, such functions you can write it as void – void swap int *ptra, int *ptrb. So, ptra and ptrb are pointers.

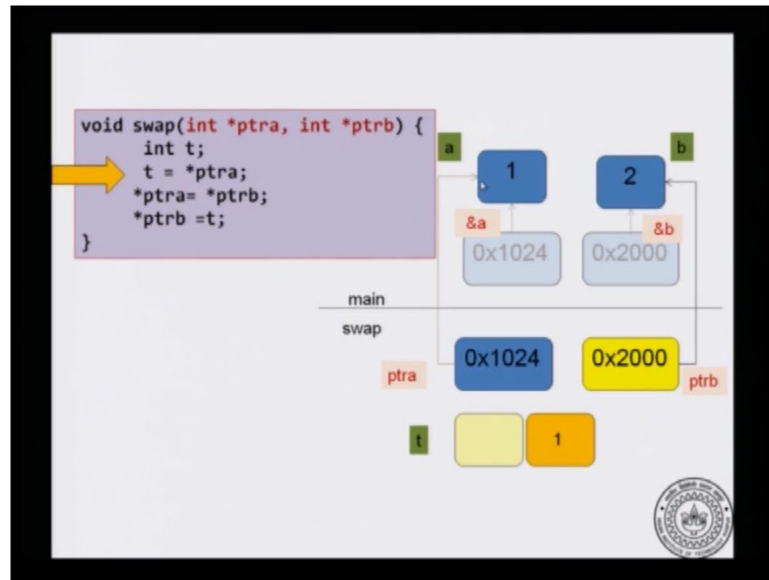
Now, inside the code, you have something that looks like a three-way exchange. It is very carefully return, because the obvious way to quote the function is not right. So, you have to be slightly careful; you have to declare an integer variable. Now, t contains *ptra; *ptra = *ptrb; and *ptrb = t. The obvious way to write it seems to be – you declare an integer *ptr t and then do this. It is not quite right; we will come to that later. So, here is the swap function. And how do you call the function? You declare two integer variables in main: a = 1 and b = 2; and then pass the addresses using &a and &b.

(Refer Slide Time: 11:27)



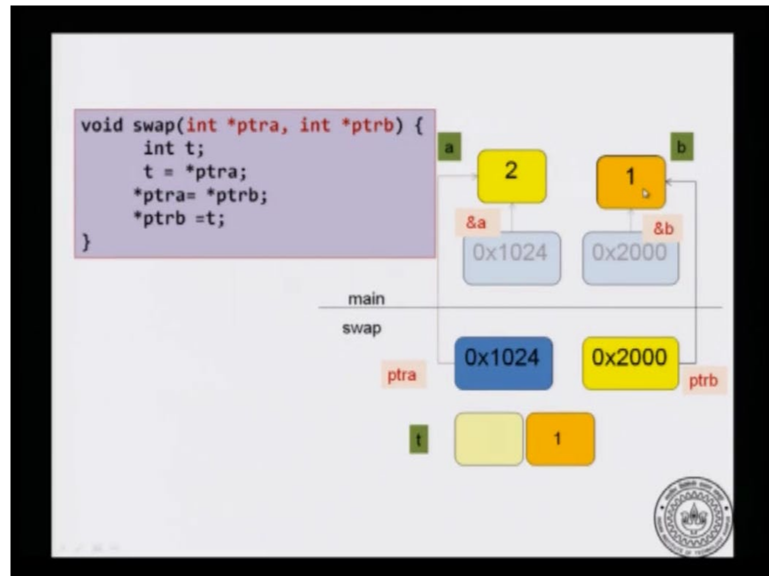
So, let us just trace the function. You have two variables in main; a = 1, b = 2; and call swap(&a and &b). Now, just to denote that, these are addresses, I will say that, these are… a is situated at location 1024 in hexadecimal. So, this is some location in memory – hexadecimal 1024. And this is some other location in memory; b is say at hexadecimal location 2000. Now, do not be distracted by the hexadecimal notation if you are uncomfortable with it; just write 1024 in an equivalent decimal notation; and you can say that, it is at that location. So, it is at that location. And I am representing the location in hexadecimal, because it leads to shorter addresses. And this is also an address. So, when I will take &a, I will get 1024x in… When I take the &b, I will get 2000x. So, this is the &a. And it is located at memory location 1024 when represented in the hexadecimal notation.

(Refer Slide Time: 12:49)



What happens when you call the swap function? So, here is the state of main. And when you call the swap function, a new bunch of memory – a new block of memory is allocated on the stack. So, first, the formal parameters are copied their values from the actual parameters. So, ptra will get &a, which is 1024; ptrb will get &b, which is 2000. Now, I declare a new variable t; t = *ptra. So, what does that mean? ptra is an address – dereference the address; which means go look up that address. So, it will go to this location and get that value. So, t will now become 1. And the next statement is somewhat mysteries; please understand it very slowly. So, on the right-hand side, you have *ptrb. This means dereference ptrb. So, we are saying ptrb is address 2000; when you dereference it, you will get the value 2. Now, where do I have to store that value 2? For that, dereference ptra. So, 1024 – dereference it; you will go to this box. That is where you have to store 2.

(Refer Slide Time: 14:19)



So, 2 will go to that location. So, what has happened due to that is that, a in name has now changed. Why? Because within the swap function, we were dealing with pointers. So, as a result of the statement *ptra = *ptrb, it has taken 2 from the main functions b and put it back into the main functions a. And that was accomplished through variables inside swap. So, think about it for a while. And the last statement of course is *ptrb = t. So, dereference ptrb and put the value 1 there. So, here is a three-way exchange that works through variables only in swap. But, since they were pointer variables, you ended up changing the locations in the main as well.

(Refer Slide Time: 15:26)

Homework ☺

Will the following code perform swap correctly?

```
void swap(int *ptra, int *ptrb) {
     int *ptrt;
     ptrt = ptra;
     ptra= ptrb;
     ptrb =ptrt;
}
```

And once you return, all the memory corresponding to swap will be erased. But then when you to main, a and b will have changed. a and b were 1 and 2 before. Now, a is 2 and b is 1. So, it has correctly swapped. Now, as an exercise, I said that, the obvious way to write the swap function is as follows. Void swap a int *ptrb; ptra and int *ptrb. And then I declare int *ptrt. And then I write these statements. This is a very obvious way to code swap. This does not work. So, try to draw these pictures as we have done with a swap function that actually worked. Try to draw the picture of what happens in main and what happens in the swap function. And understand why this particular swap function does not work.

One final word about passing pointers to functions; C has something called a call by value mechanism. What is meant by call by value is that, when you call a function, remember the original picture that, your friend came with his note book and copied down the numbers in your page. So, your friend created a separate copy of your arguments; then computed what had to be computed and returned you a value. That picture is essentially still correct. Even though you are now dealing with functions, which can manipulate memory inside main, the passing mechanism is still call by value. It is just that, what is being copied are the addresses. So, when you manipulate the addresses through dereferencing, you end up changing the location inside main. So, even with pointers in C, what happens is call by value.