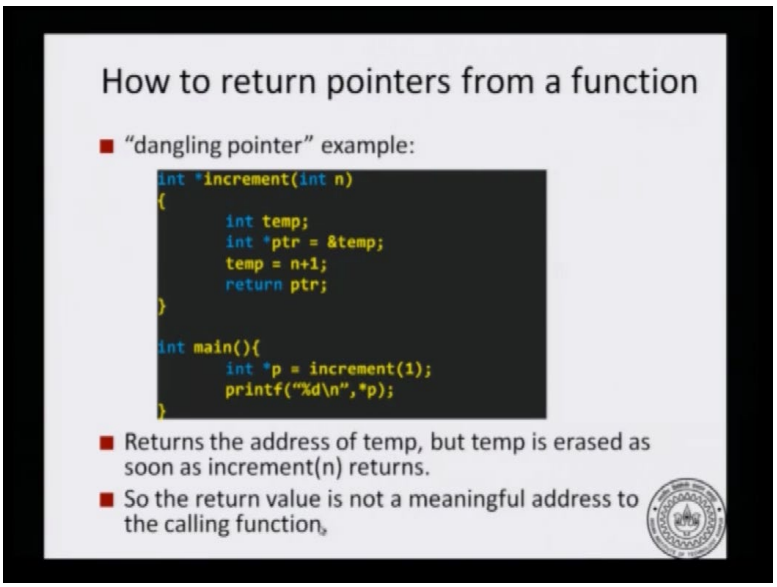
Introduction to Programming in C
Department of Computer Science and Engineering

In this video, we will discuss slightly advanced usage of pointers. Even though, the title of the first slide is, how to return pointers from a function? That is just a motivation for introducing a slightly more advance topic in pointers.

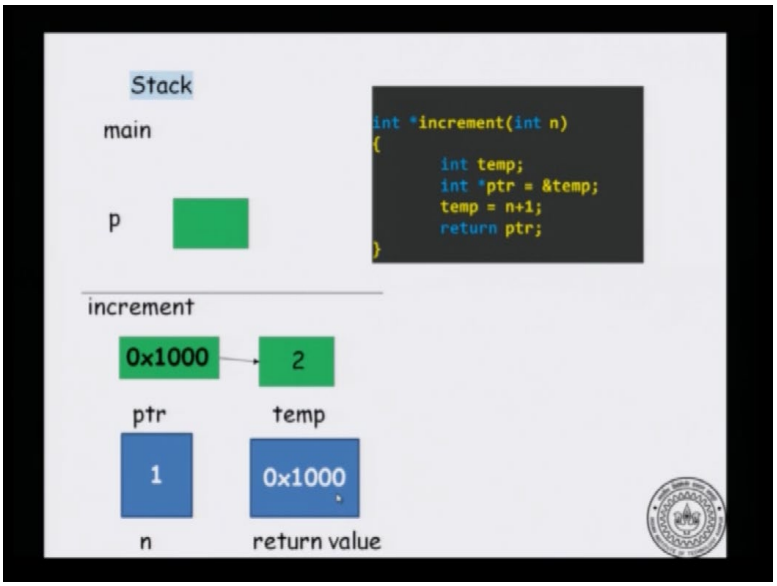
(Refer Slide Time: 00:10)



So, let us just see what is the problem with returning a pointer from a function? We know that, any variable can be passed as argument to a function, can be declared as a local variable within a function, and can also be a return from a function. So, is there something and we have already seen that, in the case of the swap function how we pass pointers to a function and what new kinds of functions does this enable us to write?

So, in this lecture let us talk about what will happen, when we return pointers from a function? I have written a very silly function here, you do not need a function to do this. But, this illustrates a point I have a main function, in which I have an integer pointer p and I will make it point to the return value of increment(1). And the increment function, what it does is, it takes an argument, it increments the value makes a pointer and points to the incremented value and returns that pointer, we will see a pictorial representation in a minute. Now, this leads to a very notorious error in c known as a dangling pointer.

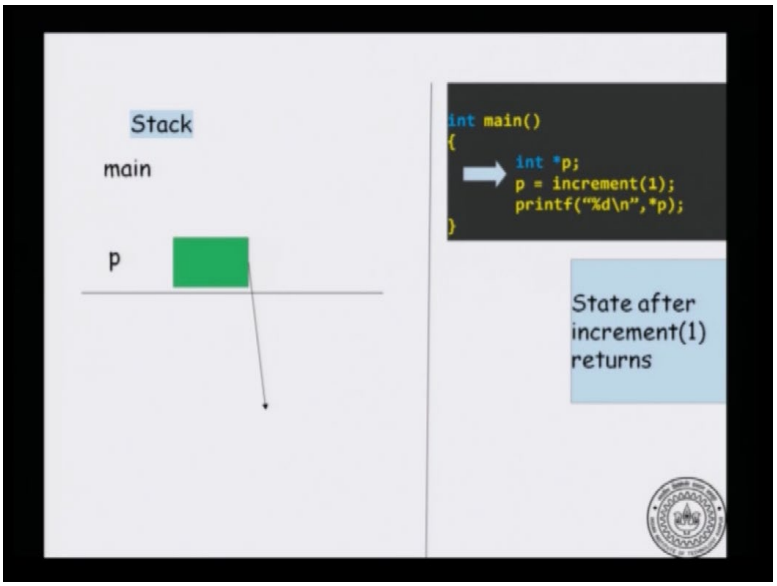
(Refer Slide Time: 01:41)



Let us examine that in slightly greater detail. What happens when we call the increment function? Inside the main function, we have an int pointer p and then it is declared, it is not pointing to anything and immediately, you will call increment(1). So, you call increment(1), n is a local variable in increment, it is the argument. So, n is 1 and then, you declare a temp and then, you declare a pointer to temp,. So, there is a ptr. Let us say that, temp is a address hexadecimal 1000. So, ptr contains 1000 and it points to temp, temp is at address 1000.

Now, in the next statement you increment temp, you set temp to n + 1,. So, n is 1 and temp is 2. Now, ptr points to temp and now, I will return ptr. So, the return value is 1000, which is the address of temp.

(Refer Slide Time: 03:01)



Now, what happens when you return to main? As soon as increment finishes and we have said this several times before, as soon as any function finishes, the memory with that is allocated to the function is erased. So, when you return to the main function, what happens is that, you have p and p will contain the address 1000. So, it is meant to point to temp. But, the space meant for temp has already been erased.

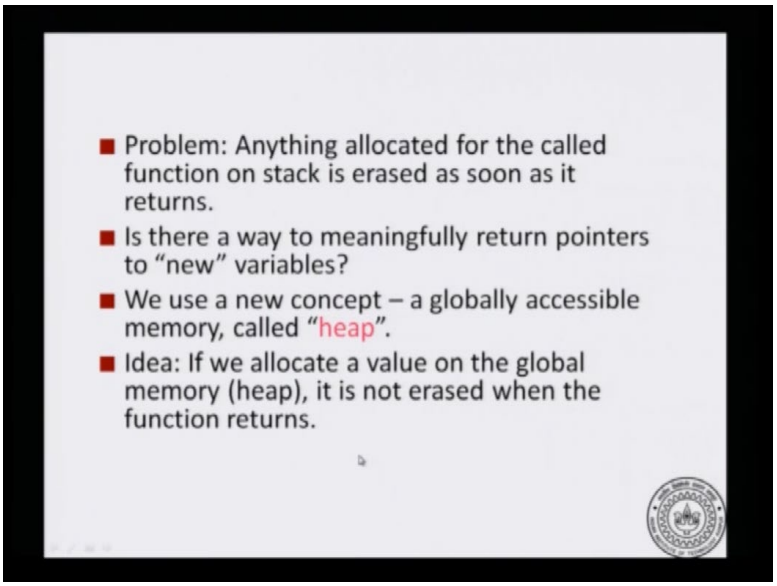
So, p is pointing to a junk value in memory, it is pointing to an arbitrary location in memory. So, this is known as a dangling pointer. Hopefully, the picture is a representative of a dangling pointer. The fact that, it points to a location, which is no longer meaningful. Notice that, what I am talking about is the ideal situation, when you code it in c and try to run it, may be p does point to the location with address two.

This is because, c may not be aggressive in re cleaning the memory. But, you should always assume that, the safe thing is to assume every location that was allocated to increment is erased immediately after increment returns. In practice, it may not be the case,, but you should never assume that, you still have the temp variable. In general, what you will have is a dangling pointer. Because, p points to a location which no longer contains any relevant information.

So, when you print p, you will have a danger. So, how do you return pointers from a function? So, we have seen what a dangling pointer means? And here is a very silly function, which will create a dangling pointer. Now, what is the problem with this function, it returns the address of a local variable temp. But, temp is erased as soon as

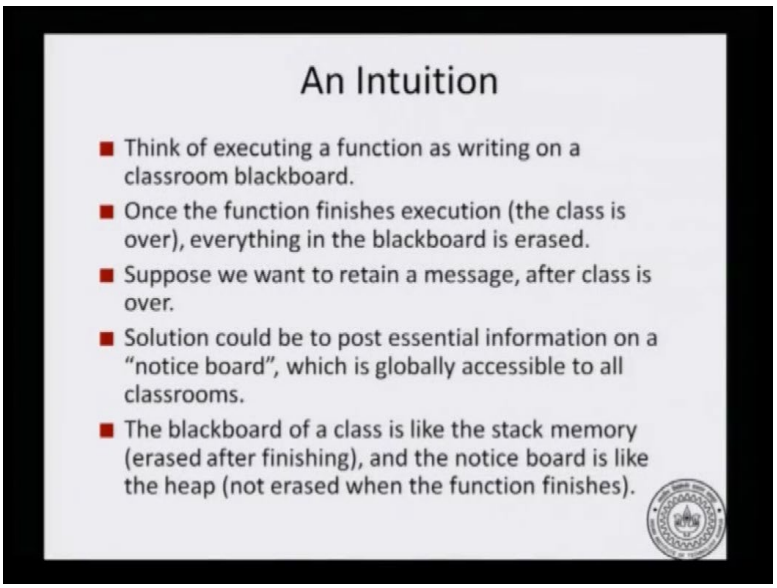
increment n returns. So, the return value is not a meaningful address to the calling function, which is mean. Can we get around this?

(Refer Slide Time: 05:11)



So, the main problem here is that, anything that is allocated to the called function on the stack is erased as soon as it returns. Is there any way at all to meaningfully return pointers to new variables? Then, use a new concept that is a globally accessible memory called heap, we have already seen a stack. Now, we will understand what a heap is. So, roughly the idea is that, if you allocate value on the global memory, it is not erased when the function returns.

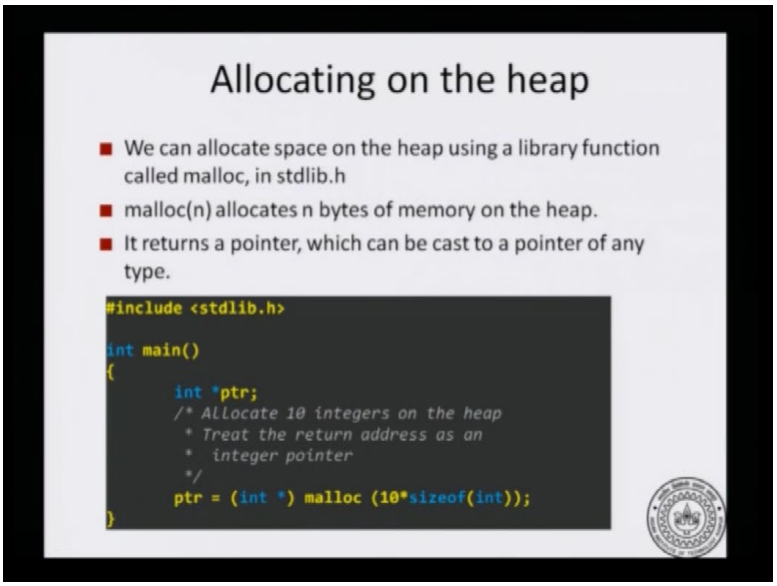
(Refer Slide Time: 05:56)



I will explain this with the help of a slightly a broad analogy. Hopefully, this is indicative of what actually happens with the heap? So, think of executing a function as writing on a classroom blackboard, when a lecture is going on. Once the function finishes execution, which is like the class is over, everything on the blackboard is erased. Suppose, you want to retain a message after the class is over. Now, the solution could be that you can post things on a notice board, which is global to all class rooms.

So, it is common to all class rooms. So, things on the notice board are not removed as soon as a class is over. If you write something on the blackboard, which is similar to storing something on the stack, as soon as the class is over, it will be erased. So, if you have something to communicate back to another class, may be you can post it on a notice board. Now, the notice board is globally accessible to all class rooms. The black board is like a stack and the global notice board is like a heap and contents on the heap is not erased, when the function finishes.

(Refer Slide Time: 07:16)



So, how do you allocate things on the heap? There is a standard library function called malloc, in the file stdlib.h which can be used to allocate space on the heap. Roughly, this is what it does, if you ask for malloc(n), there n is an positive integer, it will allocate n bytes of memory on the heap and it will return a pointer to the first location of the allocated space. Now, that pointer can be converted to pointer of any type, malloc just allocates n bytes.

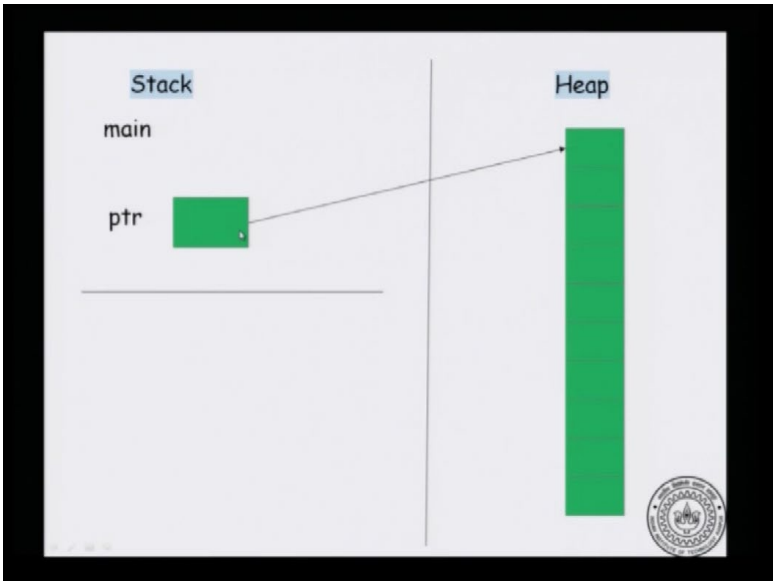
Now, you may want to interpret those bytes as n divided by 4 integers. In that case, it

will return a pointer,. So, you convert that pointer to an int pointer. Let us see an example I may have an ints pointer for ptr and now, I want to allocate 10 integers on the heap. How do you I do that? I will allocate 10 * sizeof(int). So, this will allocate on some particular machine, let us say 40 bytes and it will return an address of the first location. Now, that address I want to treat it as an integer address. So, I will convert it to an int as int * and then malloc (10*sizeof(int));.

So, this style of writing code means the code portable. Because, suppose you write the code and on a machine, where integer was 4 bytes and you take your code and go to a bigger machine, which has 8 bytes as the size of an integer. Then, you compile the code on that machine and your code will still allocate 10 integers. Why? Because, on the new machine sizeof(int) will be automatically 8. So, it will allocate 80 bytes.

So, in order to write portable code, you can use sizeof(int), instead of assuming that, integer is 4 bytes. So, I want to allocate malloc (10*sizeof(int));, this will allocate 10 integers no matter, which machine you do it all. So, and it will return you the address of the first byte in that allocated space, that address you convert to an integer array, integer pointer. Here is, how you allocate memory on the heap.

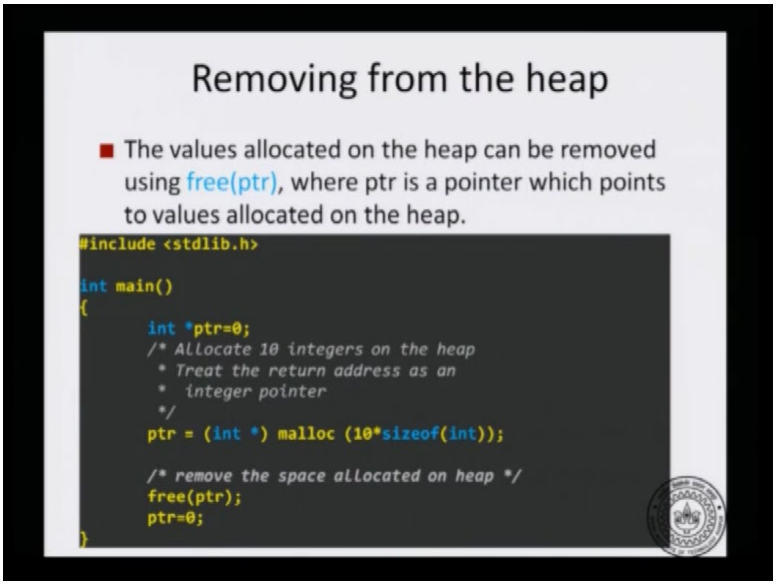
(Refer Slide Time: 10:04)



So, when we think pictorially, think of heap has a separate space in the memory. In this case, ptr will be allocated some space on the heap. Let us say 10 integers on some particular machine, it will say 40 bytes and it will return the address of the first byte. Now, that first byte you treat it as a pointer to int, that is done through the conversion int

*.

(Refer Slide Time: 10:42)



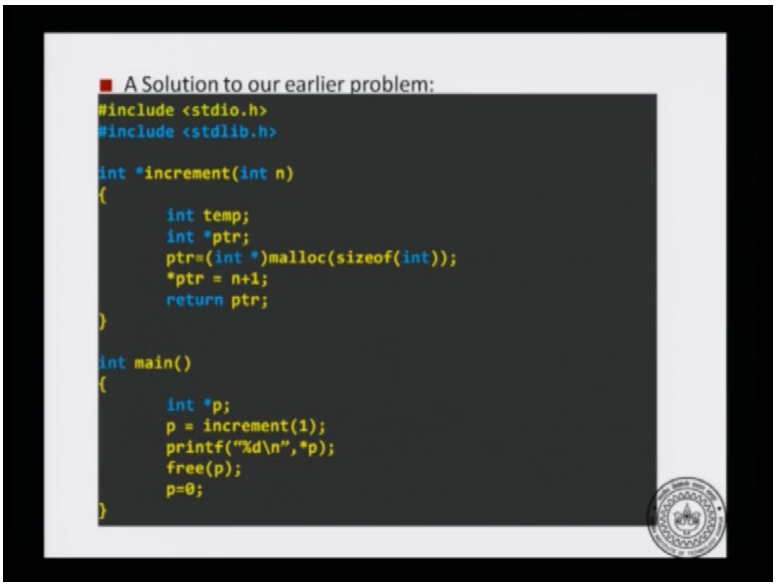
Now, it is nice that you can allocates space on the heap. But, in order to be hygienic, you should also remove the allocated space, once you have done with it. There should be a reverse operation to allocate and that is free, it is in the same library, stdlib.h. And if I just say free(ptr) and ptr was originally allocated using malloc. Then, it will correctly remove,; however, many bytes were originally allocated.

So, let us say that I have int *ptr and then ptr, I allocate 10 integers on the heap and ptr is the address of the first allocated location. Now, I may do a bunch of processing here and once I have done, it is just nicer me to de allocate things on the heap. This is like, saying that things on the notice board once some condition occurs, where you know that the notices no longer needed, you just remove that posting from the notice board for that, we use free of ptr.

Now, notice the asymmetry here, malloc took the number of bytes to be allocated free just wanted to say, which pointer is to be free? It does not ask for, how many bytes to free? So, you can imagine that malloc does some kind of book keeping, where it says that I allocated 40 bytes and that was return to ptr. So, if I just say free(ptr) it automatically knows that, 40 bytes are to be free, you do not have to give the extra argument saying, how many bytes to free?

Once you free the pointer, you just set it back to null, this is just a safe practice and it is not absolutely necessary,, but it is recommended.

(Refer Slide Time: 12:57)

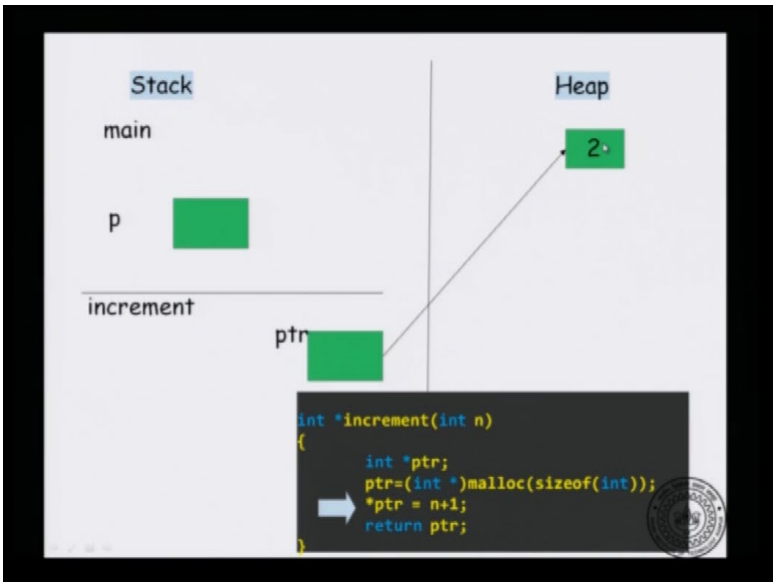


So, let us solve our earlier problem using malloc. Our earlier problem was that, ptr was pointing to some location within the stack. So, as soon as the function returned, the return address no longer meant any meaningful address. So, let us now solve this problem. I have included stdlib.h, because I will allocate memory on the heap. So, the increment function is modified as follows strictly speaking, I do not need a temp variable any more.

I have an int pointer and I will use the pointer to allocate one integer on the heap, this is a really wasteful practice but, it just illustrates a point. So, it will allocate one integer on the heap and then, return that address and treat that address as int *. Now, I will use * ptr = n + 1 to dereference that location on the heap and set the value to n + 1. Once I am done, I will return the ptr I will return the address on the heap.

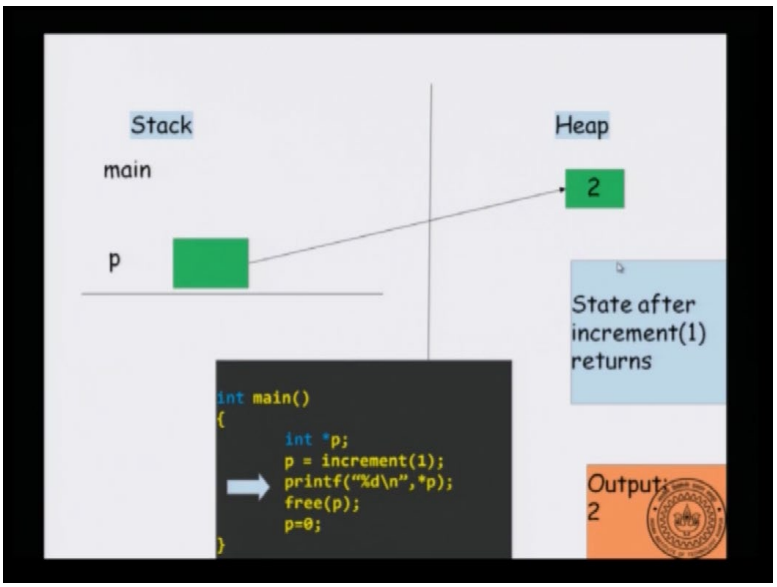
There are differences here is that, their return address is on the heap and only the stack is erased. So, things on the heap are not erased, unless you explicitly ask it to be erased, via free. So, returning a heap address and,. So, p will point to a meaningful location on the heap, when you print it, you will get too. And once you have done, you can say free(p). So, here is a strange use which you have not seen before something was done. The malloc was done in the increment function and the free is being done in the main function. Now, if you think back to the physical analogy, it is not really surprising somebody can post something on the notice board and a different person can remove it.

(Refer Slide Time: 15:12)



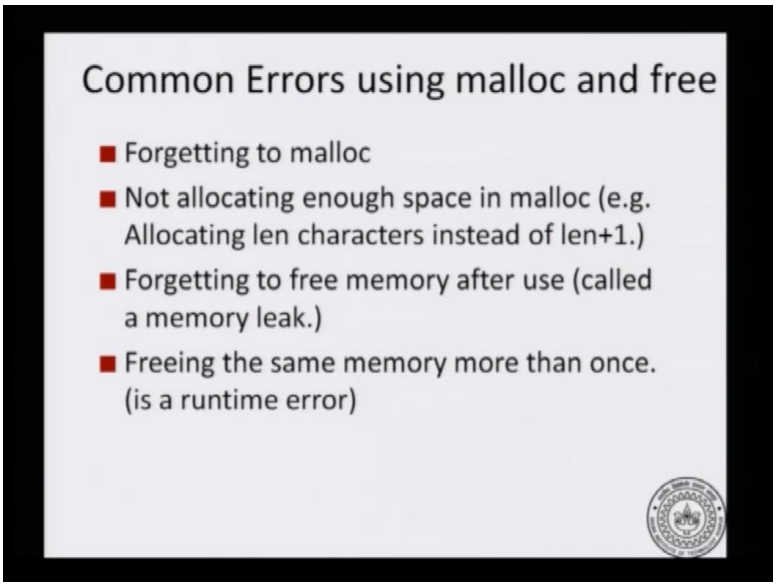
What happens here is that, the increment function ptr points to some location on the heap, using malloc. So, one integer is allocated on the heap and when you say, *ptr = n + 1's, then the location in the heap will contain 2.

(Refer Slide Time: 15:34)



And here is the catch, earlier p was just dangling, it was just pointing to an arbitrary location in the memory. But, increment allocated something on the heap and returned that address. As soon as increment returns, the stack is a waste. So, everything that was allocated on the stack for increment is erased,, but things that are allocated on the heap remain. So, p points to a meaningful address on the heap, then once you are done you can say free(p) and things will be erased, when you print it, the output will be 2.

(Refer Slide Time: 16:22)



Malloc and free are prone to a lot of errors and a lot of programming errors in c, can be trace back to incorrect use of malloc and free. So, there are some categories of errors for example, you may forget to malloc in the first place. So, you will lead to dangling references or dangling pointers, as we saw in the first example. Now, you could allocate some space. But, you may not allocate enough space, that is a very common error.

Commonly, you could allocate of by one errors I wanted to allocate really len + 1 number of bytes. But, instead I allocated only len number of bytes. Another very common error is something known as a memory leak, which is that you allocate things on the heap,, but you forget to free memory after use, this is called a memory leak. Notice that, if you allocate space on the stack, it will always be cleaned up as soon as the function returns.

So, memory leaks usually happen, when you malloc space on the heap. But, you forget to free them, once you have done and a lot of software ships with memory leaks and this is a major concern in the industry. This is also an obscure error, which is freeing the same memory more than once. This is uncommon when a single programmer is working on a code. But, when multiple programmers are working on the same piece of code, you may end up freeing the same memory twice, this will lead to some run time errors.