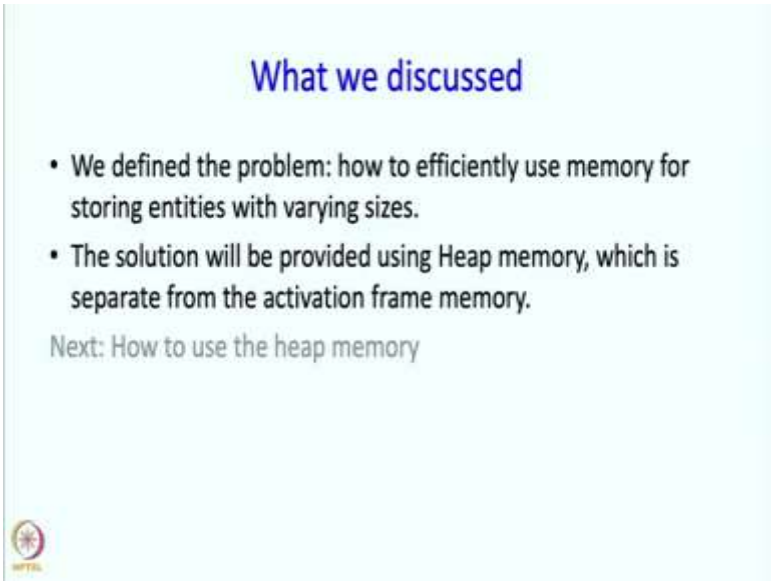
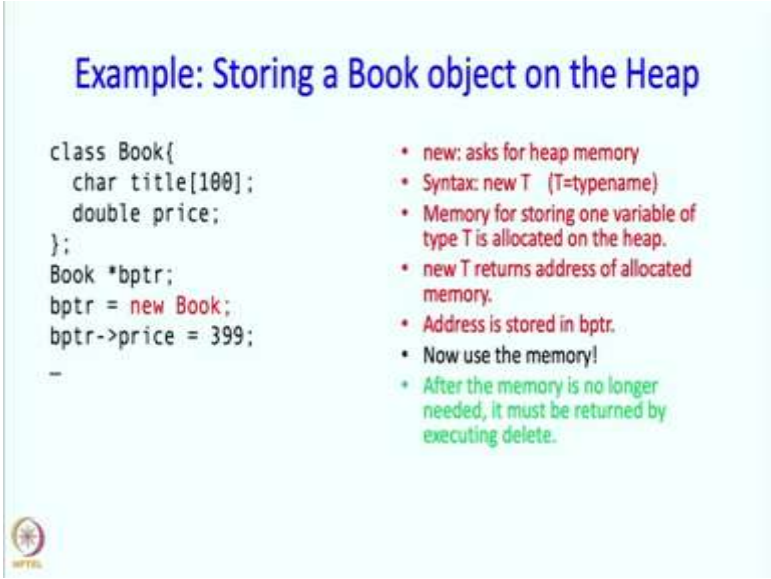
An Introduction to Programming Through C++
Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture-22 Part-02
Representing variable length entities
Heap memory basics

(Refer Slide Time: 00:20)



Welcome back. In the last segment we started off by defining a problem that we wanted to solve which is how to efficiently use memory for storing entities with varying sizes. And we said that there is something called heap memory, which can be used to implement solutions for such problems. In this segment we are going to use this heap memory, we are going to see how to use this heap memory.

(Refer Slide Time: 00:47)



So, let us take an example, we want to store a book object on the heap, so here is our book object, so it is a class book, char title and there is price, maybe I should call it struct because I am using the internals the members directly, so perhaps I should called it struct but anyway. So, I can declare a bptr to be a pointer to a book object.

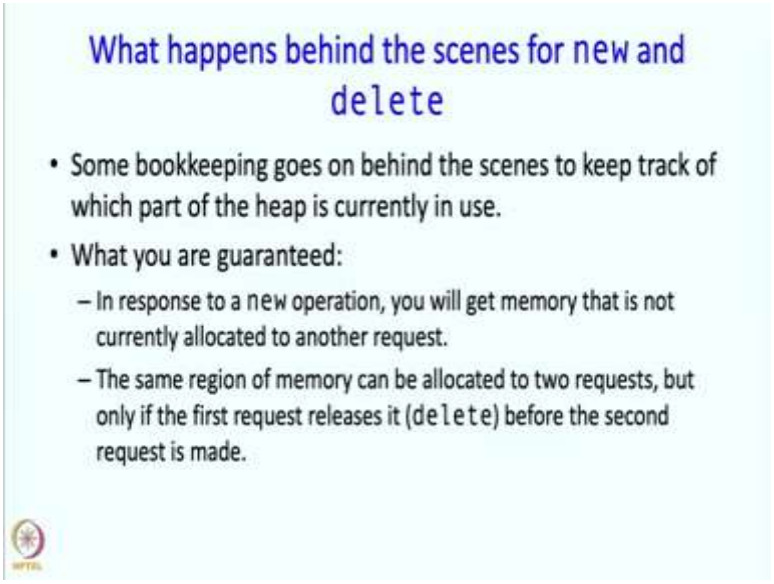
So, right now bptr is not does not contain a value, I am just defining that variable. Now, here I am assigning a value to bptr and this is a new statement, so this says bptr equal to new book, new is actually an operator and this operator asks for heap memory. The syntax in general is going to be new followed by the name of a type. And what this causes is that memory for storing one variable of type T is allocated on the heap, is reserved on the heap and the statement new T or new book returns the address of the allocated memory. So, the address of that allocated memory comes back to the program and then that address gets stored in bptr.

So, bptr now has a pointer or bptr contains the address of some location in the heap memory which has been given in response to this request. So, now, we can use that location so we can write bptr->price because bptr is a pointer, so we go to that variable and we look at the price member from it. And in this way we can keep using the memory as much as we want, we can change the price, we can change the title, we can do whatever with it.

At some point, we will discover or we will decide that look I do not need this memory any longer. So, in that case, we should return the memory back, we should tell the heap or the heap

manager so to say, that look I was using this memory but now I do not need it so take it back and maybe give it to me later if I ask for it again. So, that is done by this command delete bptr, that returns that memory back. New and delete are reserved words and they are also operators. Well a lot happens behind the scenes for new and delete.

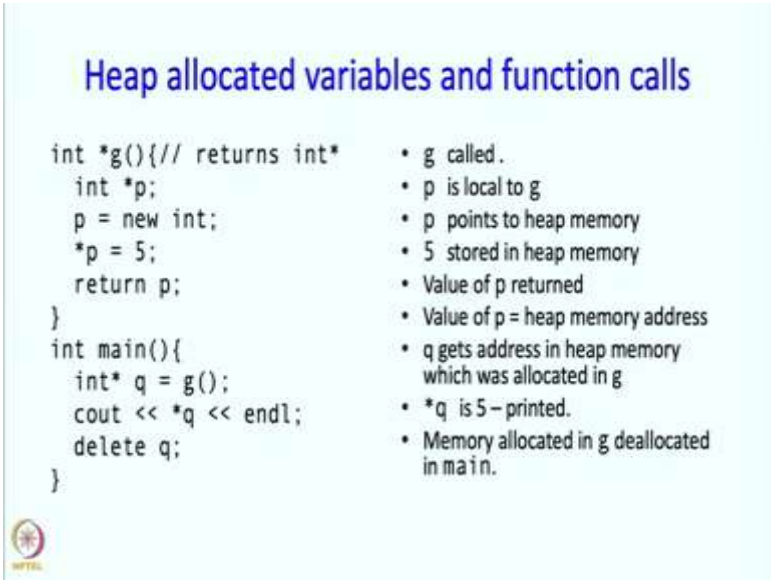
(Refer Slide Time: 03:41)



So, there is some bookkeeping which keeps track of which part of the heap currently in use. And as a result of this bookkeeping you are guaranteed that in response to a new request you will get memory that is not currently allocated to another request. Of course, if the heap is full then you will run out of memory and your program will abort. But if it is not full then you will get that memory.

And the same region of memory can be allocated to two requests but only if the first request releases it using the delete before the second request is made. So, at the same time, the same memory cannot be in use due to two requests. So, here is an example showing how the heap memory functions and how it is different from standard activation frame memory.

(Refer Slide Time: 04:38)



So, in this program let us look at main, so main starts off with creating a variable q which is a pointer to int, which is of type int star and q is being assigned the result of calling g. So, g is called and noticed that the g is actually returning a pointer to int, int *g. So, int * is the return type and g is returning a pointer to int.

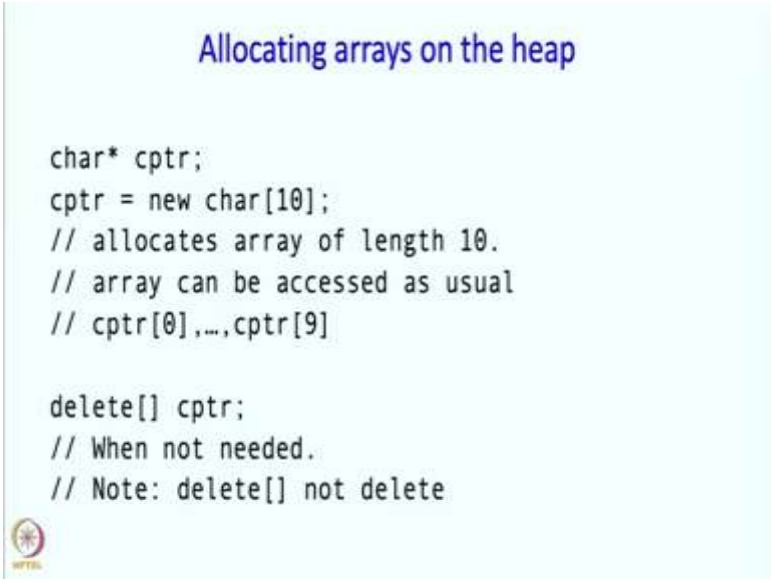
So, what happens inside g? Well first a local variable or p is created of type int *, so this is in the activation frame of g, then we execute our new statement p equals new int, so what does this do? Well it places an address in the heap so it causes a location in the heap to be allocated and that address is placed in p. So, p is now pointing to some location in the heap memory which has been given for it. Then we say *p=5, so what does this do? Well *p takes you to the location in heap memory and you place 5 over there. So, 5 is stored in the location in the heap memory which was given to us and now we are returning p. So, what does return p mean?

Well whatever the value of p is returned, but what is the value of p? So, the value of p is the heap memory address, so this heap memory address is returned. And this goes and takes the place effectively of this call and it sits in this q, so that address goes and sits in q. And now when we print *q, what is happening? So, q at this point contains an address in the heap memory which was allocated over here. So, *q is, q is pointing to that same address but into which we had stored a 5 over here. . And therefore if you print *q it will print 5. So, at this point we have

discovered that we do not need q any longer or we have decided that we do not need q any longer and therefore we are going to delete q.

So, notice that there is something unusual going on from the point of you of what you have known so far which is that some memory got allocated over here and it stayed across this function call boundary and then it was used in the main program and then it was deleted in the main program. So, this is clearly very very different from the way the memory allocation happens on the activation frames.

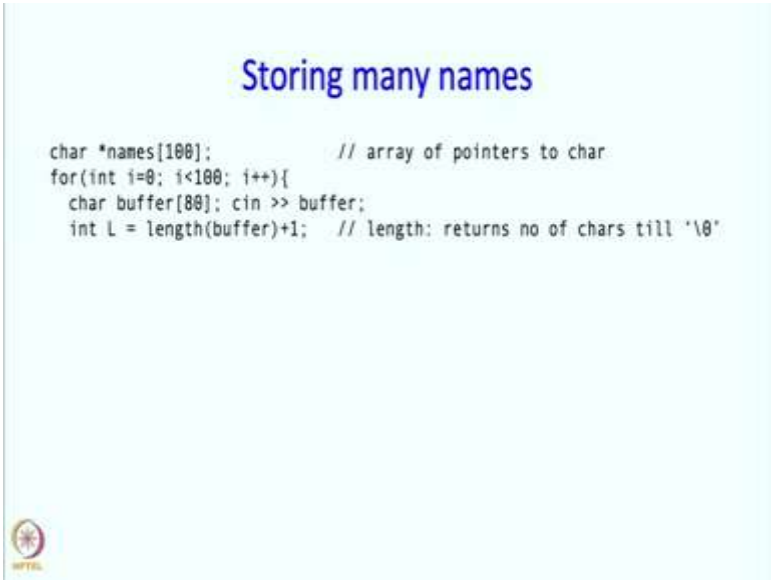
(Refer Slide Time: 07:43)

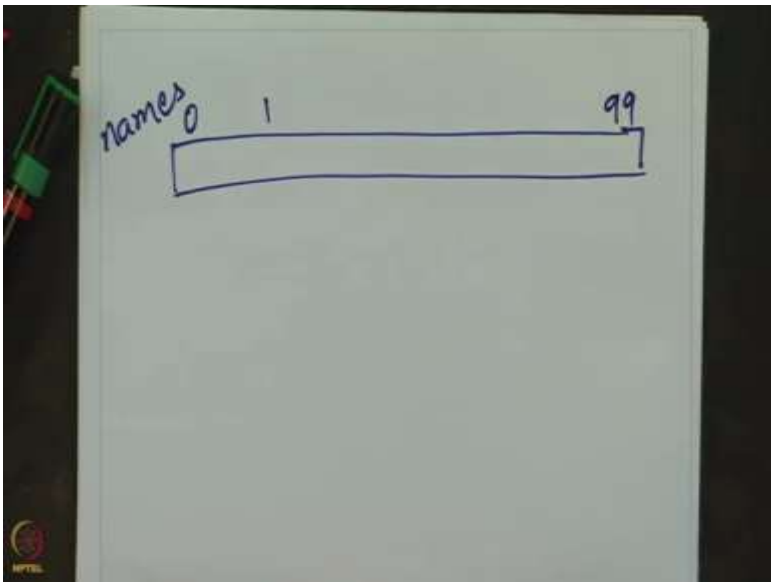


You can allocate arrays on the heap and so again let us say cptr is a pointer to characters. Now, I can say cptr equals new char 10. So, this will give me an array of length 10, an array of char. And now I can start accessing the array as usual because after all the name of the array is an address, address of the starting and cptr also is an address, address of the starting point of that array, so I can get the usual variable cptr[0], cptr[9]. So, the array elements I can just access in this manner.

After I am done with the heap I can write delete[]cptr, so this causes the allocated array to be deleted. So, note that it is not just plain delete but it is delete[]. Alright so now we have pretty much, we are pretty much in control of the situation so our problem was storing many names and now we are able to do so. So, let us see how.

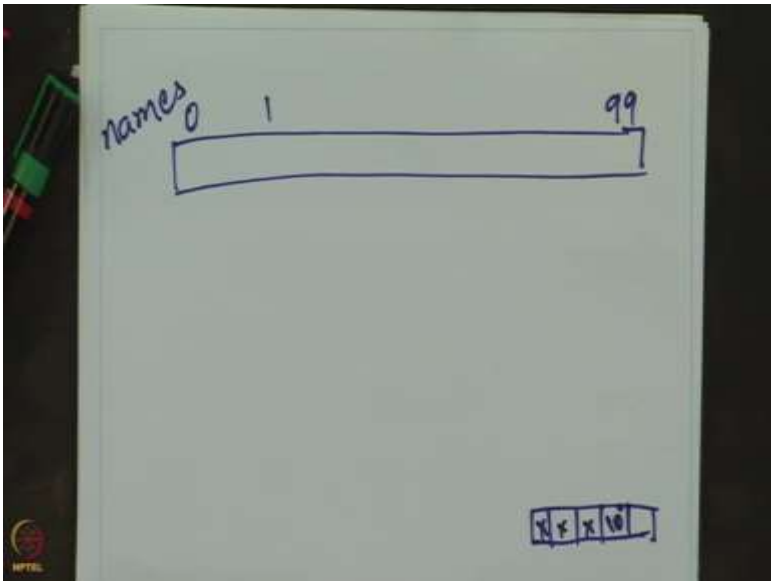
(Refer Slide Time: 09:05)

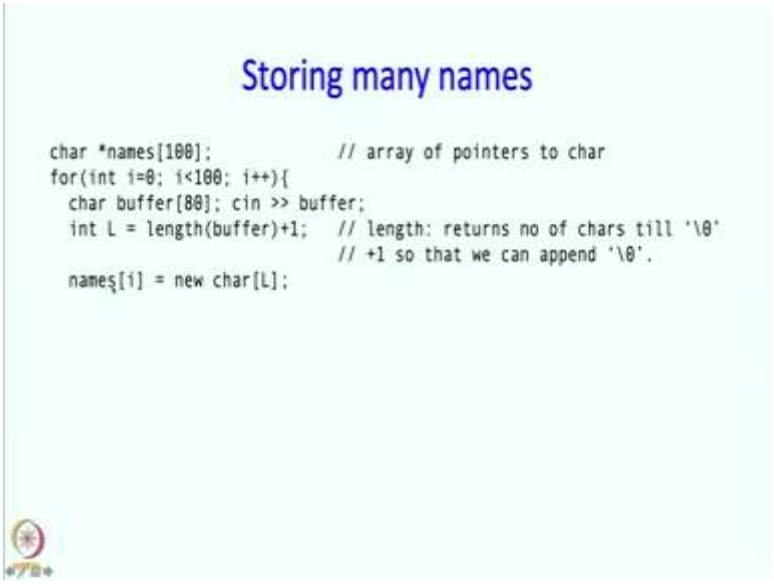




So, let us say we want to store 100 names and so I am going to define an array of pointers, pointers to char. So, I have an array of pointers and initially I have not put anything in it. So, let me draw it pictorially, so this is my array names and let us say this is 0th element, first element all the way till the ninety ninth element. So, what happens next? So, I am going to go over all the elements of this array and I am going to have an iteration corresponding to all the elements of array and in the iteration, I am going to read in something into the buffer, I am going to read in the names into the buffer.

(Refer Slide Time: 10:08)



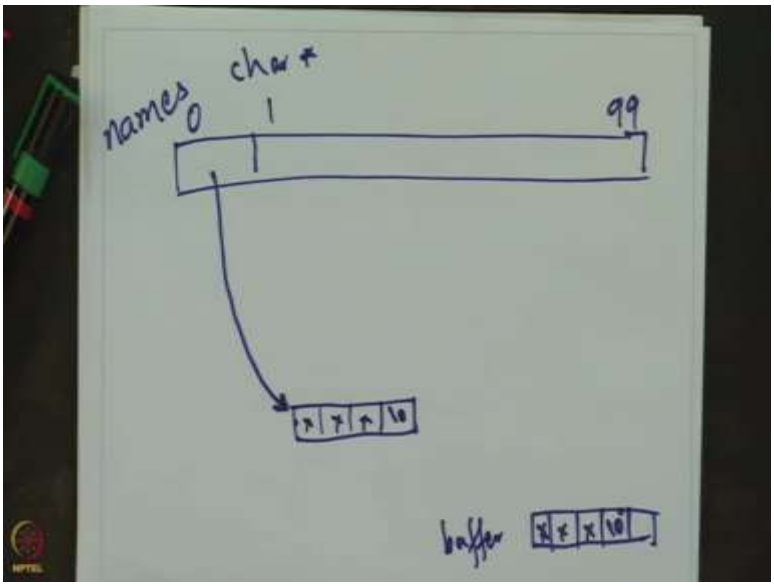


Then I am going to calculate the length, so say on the side over here I have this buffer into which I read in something and as usual there will be a null after that. And now I am going to call a function “length” which is supposed to return the number of characters till the null. So, in this case it should return 3.

So, this function we have discussed earlier or we have discussed something like this earlier and this function is also discussed in the book, but it is a very simple function so you should be able to write it very very easily. So, it just returns how many characters there are until the null. And L is set equal to that length plus 1.

So, what is the idea here? So, we want to move, we want to allocate an array of just the right size, so L is going to be just the right size, because in that array we want to store the name that we just read but we want to append a null character to it. So, the plus 1 is so that we can append the null.

(Refer Slide Time: 11:28)



Storing many names

```
char *names[100];            // array of pointers to char
for(int i=0; i<100; i++){
  char buffer[80]; cin >> buffer;
  int L = length(buffer)+1;  // length: returns no of chars till '\0'
                             // +1 so that we can append '\0'.
  names[i] = new char[L];
  for(int j=0;j<L;j++)
    names[i][j] = buffer[j];
}
```

- The jth character of the ith name can be accessed by writing names[i][j] as you might expect.

And now we are going to allocate space. So, names[i] so in general let us start form 0 so this is a pointer. So, this is of type char *, so it is going to be pointer to a character. So, we are going to do in this statement over here, we are going to do names[i] or say names of 0 in this zeroth iteration equal to new char L. So, there is some heap somewhere over here and from that heap I

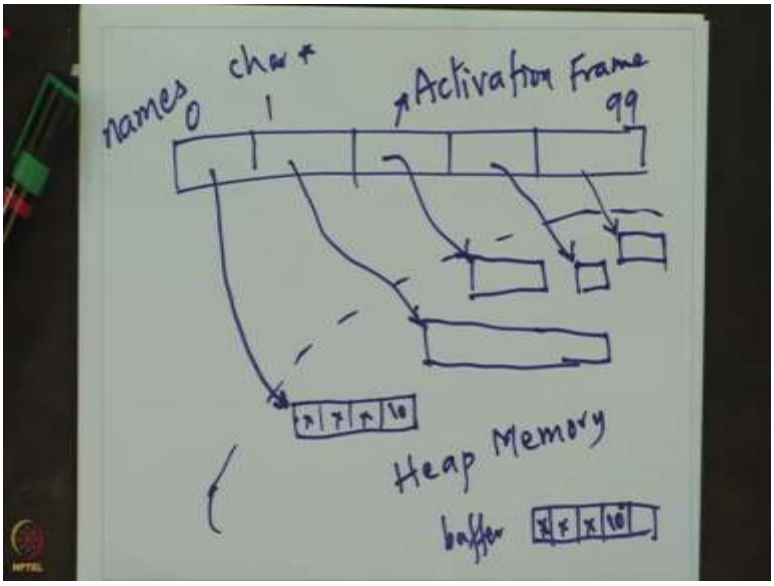
am going to get an allocation of an array of say in this case if L is 4 because there are 3 characters plus one then I am going to get an allocation of size 4 and the address of this is going to be put in over here.

So, typically we do this, draw this by making an arrow saying that whatever is here is pointing to this. So, it really says that contained over here is the address of this location, so we have just allocated space for storing this in the heap and we have a pointer to that from our names array.

Now, we simply are going to copy, so we are going to copy from the buffer, so this is our buffer, so from this we copy into this. So, whatever these three letters were followed by a null is going to get copied over here. So, this is our buffer and this is names[0][0], names[0][1], names[0][2], names[0][3].

So, what is happened over here is that we read a name, we allocated space for it plus 1 so that we want to pad in the null and then we copied that into the name into the newly allocated space and this we are going to do for all the 100 names that we want to work with. So, that is it, so that is the program. And so I guess it is a good idea to see what the final result is going to look like.

(Refer Slide Time: 13:36)

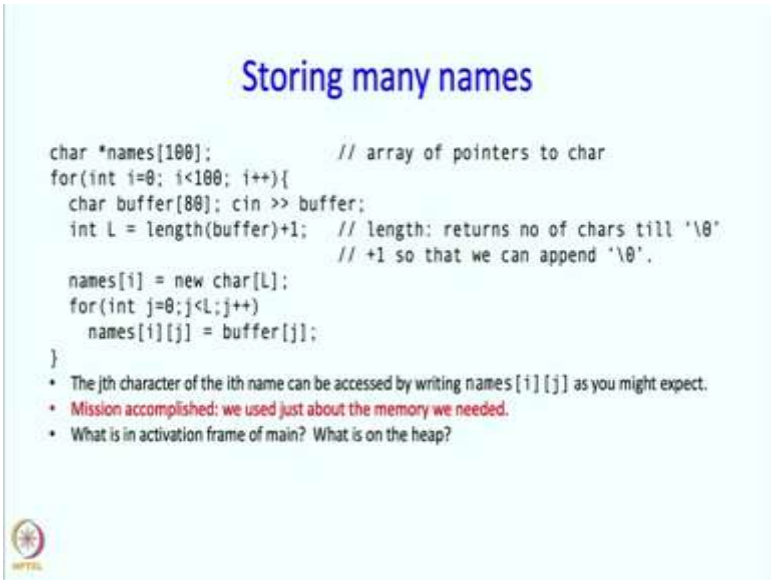


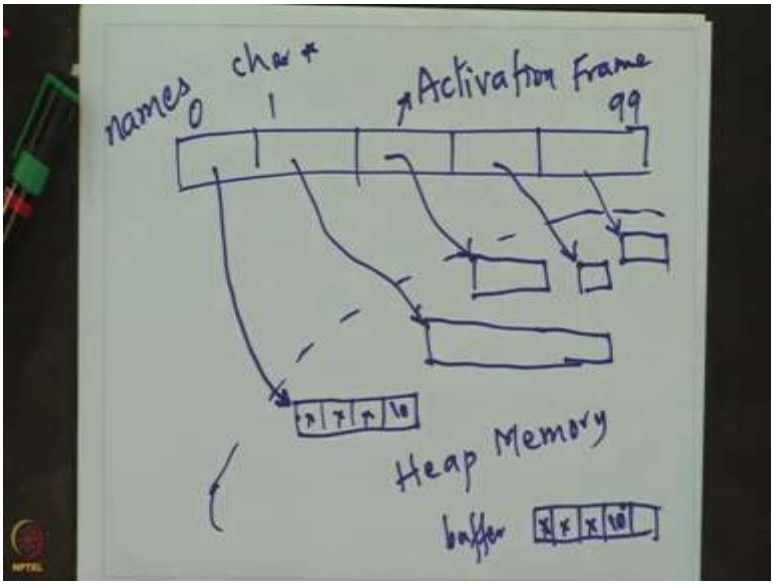
So, this names array, so this is going to be present in the activation frame. So, if this is the main program then this array is going to present in the activation frame of the main program. And this however is present in the heap memory and similarly from here another array there will be a

pointer to another array maybe this time the name is long. From here there will be another pointer to another array and so on.

So, from each element of this names array, there will be a pointers to arrays in the heap memory. So all of this part is stored in the heap memory. Now, in this case I have not shown you the deletion but of course you can write a loop and again once you are done, you can delete, delete all the variables and give back the memory to the heap.

(Refer Slide Time: 14:43)

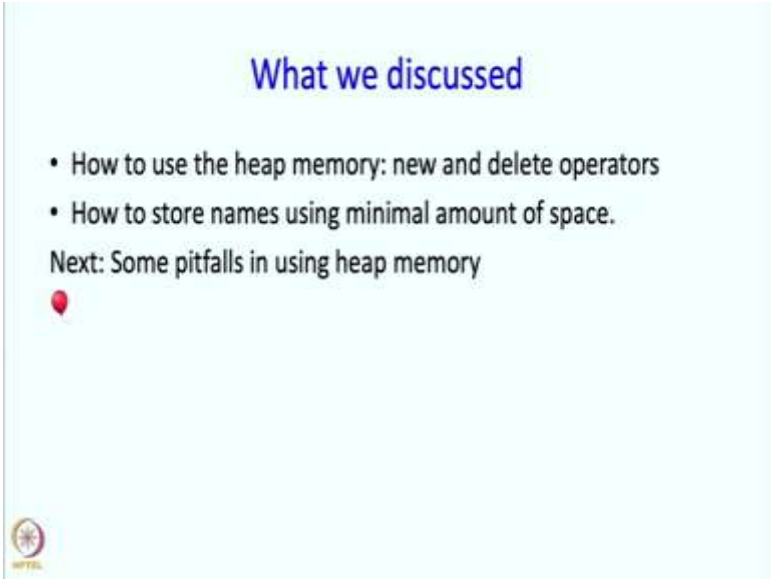




So, this is the solution that you were asking for and we have just about used the memory that we needed, we used one null character extra for each name. So, that is very minimal. And what is in the activation frame of main and what is on the heap we already discussed that.

So, in the heap memory all the arrays storing the names are there and in the activation frame the array names itself which is, which contains the pointers is in the activation frame.

(Refer Slide Time: 15:22)



Alright, so what have we discussed in this segment? So, in this segment we talked about how to use the heap memory. In particular, we have talked about new and delete operators. And then we talked about a solution to our the problem with which we started the lecture and we showed that using new and delete, we can in fact store all the names in reasonably small amount of space and then we can get to those names quiet easily as well.

Now, it turns out that while this solution is acceptable, it is actually quiet tricky to use the new and delete operators and so we need to really evolve something more than just these two operators, we need to have some scheme, some policy for using new and delete and there are some difficulties in it, some pitfalls. And in the next lecture we are going to talk about these pitfalls but before that let us take a short break.