

Introduction to Problem Solving and Programming

Prof. Deepak Gupta

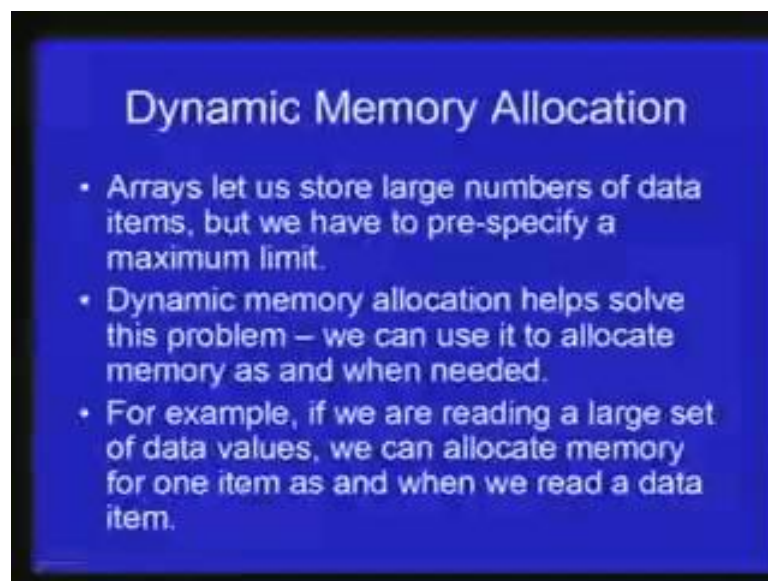
Department of Computer Science Engineering

Indian Institute of Technology, Kanpur

Lecture No. # 24

In the last lecture, we had talked about structures, and also towards the end of the lecture, we saw how we can write programs that read data from files and write data to files. Today, we are going to talk about another very important tool in programming, and constructing interesting data structures for holding various kinds of data, and that dynamic memory allocation and link list.

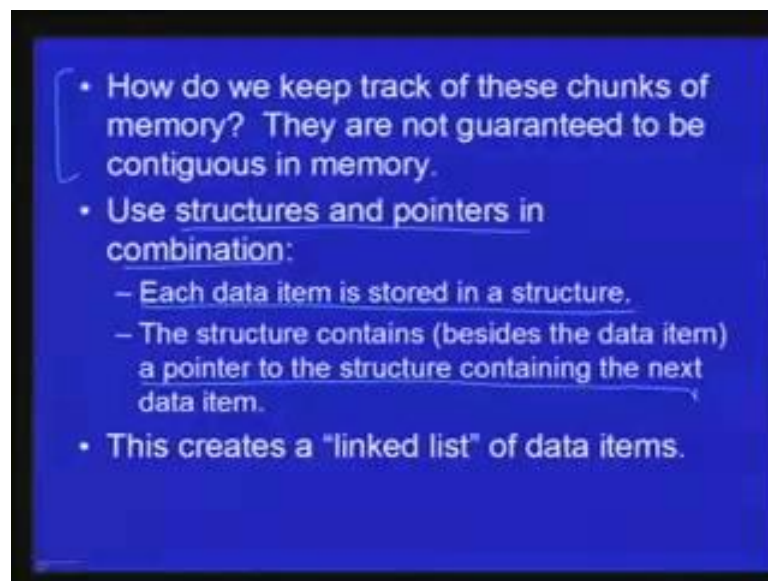
(Refer Slide time: 00:50)



So, to motivate we use of dynamic memory allocation, let us see what the problem in using arrays for storing large volumes of data are arrays let us store large numbers of data items, but as we know we have to pre-specify a maximum limit. This maximum limit has to be declared at compile time, and we cannot change that run time. And if we have more data items at run time as input, than the specified limit then we cannot really handle that. And we specify a very large limit, and the problem is at we might end up

wasting a lot of memory. So, dynamic memory allocation as the name suggest allow us to ask the system for more memory as, **(())** required.

So, we can essentially not allocate all memory at compile time itself, instead depending on the input as and when more data items are needed to be stored, we can ask for more memory. For example, suppose we are reading a large set of data values, we can allocate memory for one item as and when we read a data item, we do not even meet to know any advance found data items for going to be the error.

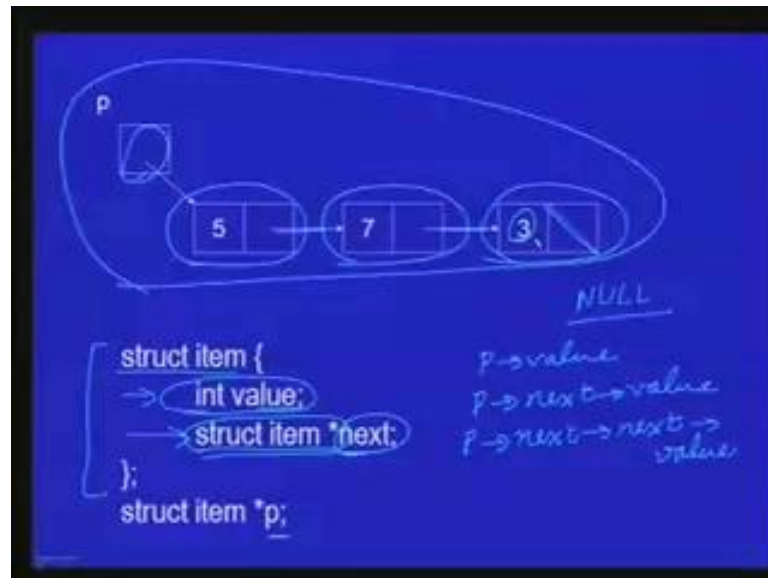


But, the problem is dynamic memory allocation is that we need to keep track of all these chunks of memory, that we get dynamically allocated and take of an array all the elements of the array are at contiguous memory location. So, if you know the address of the first element of the array, then it is easy to find the address of any array element as we know using **(())** or using the **(())** mechanism for array.

But, when we dynamically allocate large number of chunks of memory one after the other, they are not guaranteed to be **(())** memory; and therefore, we have to somehow keep track of where these chunks of memory are in memory. And the solution to be particular problem is to use structures and pointers in combination. And basic idea is very simple, that say we have integer as data items, which we need to store, and we are using dynamic memory allocation to store them and allocate, enough memory to hold an integer as and when we read a integer.

So, what we do is that each data item is stored in a structure, which besides containing the integer or whatever the type of data items is, it will also contain a pointer **(())** structure containing the next data item. And so every structure will contain a pointer to be next structure containing the next data item, and that one will contain a pointer to the third structure and so on, so forth.

(Refer Slide Time: 03:30)



And we can see this creates a linked list of data items, let us feel better to a pictorial representation. So, in this example again we are assuming that it is integer value, that we are interested in storing, so we have declared structure type, struct item which consist of an integer value and a pointer to another struct item. So, more that in **(())** strut item I cannot have ever strut item, because then the **(())** would have to been infinite, but I can always store a pointer to a similar kind of structure will been a structure itself.

These kind of structures are called recursive structures and that is possible, because the size of a pointer is independent of the size of the quantitative **(())** point. So, the compiler knows at compile time that an integer will take, so many bytes and storing a pointer will require so many bytes, and therefore overall the structure will requires, so many bytes.

And so we have the structure which computes of an integer value that **in** in to store and a pointed to be next structure, and let us say we have a variable p which is a pointer to an item. Now, consider both kind of a linked list, so the variable p contains the pointer to be first element in the list each of this is an element, and each of this is a file strut item. So,

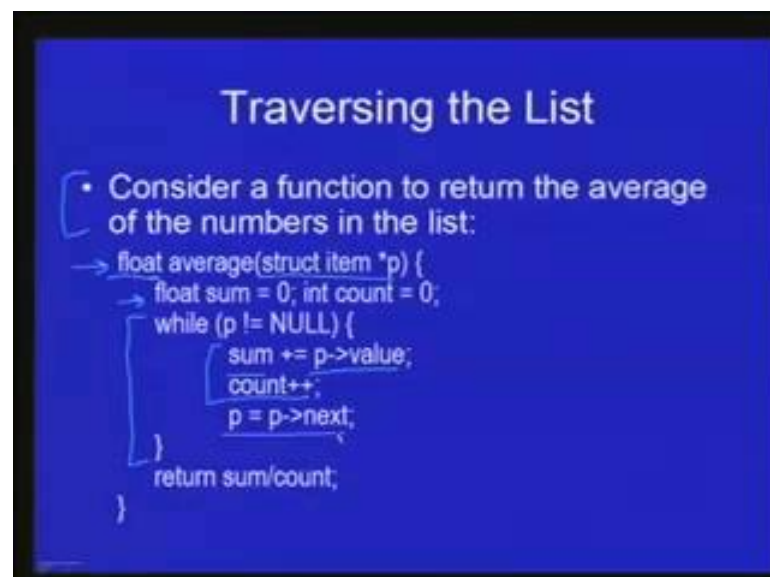
p contains a pointer to be first element and the next field of the first element contains a pointer to the second element, which has the value and pointer to be next element, which again contains 3.

And the pointer value in the next element is a null pointer, which we have already talked about earlier **in the**, in some earlier lecture, special pointer value which is not be address of any legal memory location. So, we can used this to mark the under the list, but **we have to** we have to be careful that we never dereference the null pointer, that is we never perform a share operation or narrow operation on a pointer which is null.

So, if we have this kind of a list, then we need to store only **in a** in a variable, we need to only store the starting address of the list or the pointer to be first element of a list, because the other elements can be track using these chain of pointer. So, p is a pointed to be first element, so p arrow next will be a pointer to the second element, and similarly p arrow next arrow, next will be a pointed to the third element and so on so forth.

In similarly, p arrow value is data value of the first element, p arrow next arrow value is the data value, which second element, p arrow next arrow next arrow value is the data value of the third element in the list which is 3 in this particular case. Now, before we see, how you can actually create a list, let us see an example, where we actually go through this list and compute some quantity or look for something or **or** something of that kind.

(Refer Slide Time: 06:22)



Traversing the List

- Consider a function to return the average of the numbers in the list:

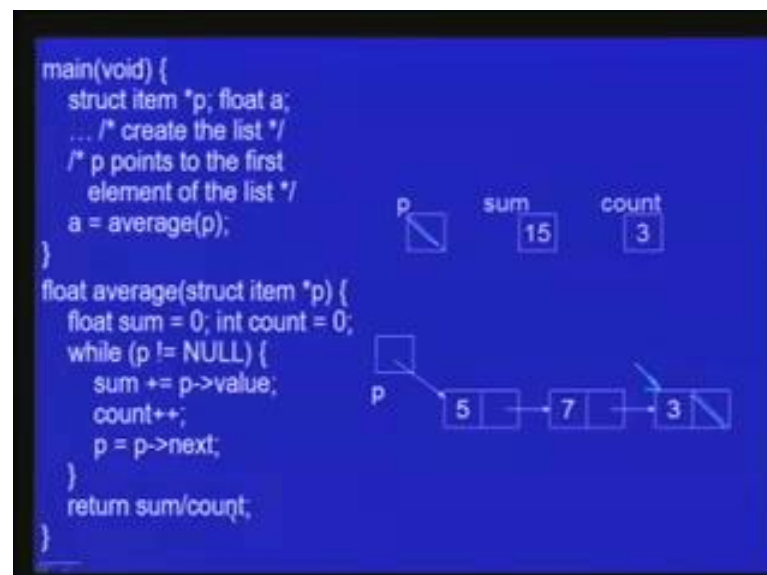
```
float average(struct item *p) {  
    float sum = 0; int count = 0;  
    while (p != NULL) {  
        sum += p->value;  
        count++;  
        p = p->next;  
    }  
    return sum/count;  
}
```

So, as an example, let us **(())** we have such a list and we want to write a function, which computes the average of the numbers in this list and of course, it assumes that the number of element is at least one. So, this is the function prototype, we going to written a float, which is the average. The argument is a p, which is of type struct item star, so it the argument is supposes to be a pointer to be first element of the list, for which you want to compute the average.

So, we have two local variables sum will represents the sum of the data values in the list, and counting to represents the number of elements in the list. Now, this loop essentially traverse in the list as long as p is not null, we in every step to make p point to be next element in the list, and the element it was counting to we add the data value to sum and increment count by 1.

So, as long as p is not null, **sum** p arrow value is added to sum count is incremented by 1, and then p is next point to be next element of the list and finally, p will become null because, the last element of the list will have the next value null, at that time which loop will terminate. Then sum will be the sum of all the element in the list, count will be the number of element in the list. And therefore, sum divided by count will be the average.

(Refer Slide Time: 08:06)



So, let us see with an example of this execute, so in this example, in the name function we have somehow created the list, which we have an seen, so far will look at that shortly. And let us assume that p points to the first element of the list, next again assume that this

is what the list look like. So, this variable p is the, variable p in mean, now when the function average is called as you know `(())` going to be created for loop the parameter p of average and the variable from and count which are local variable this average. And `the` the parameter p will be initialize the value of the argument which is this p; and therefore, that will be a pointed to be first element of the listed here. So far p is greater to the value of this p which is the pointed the first element, so this p also gets a pointer to be first element in the list.

Now, let us go through the loop p is not null and so therefore, this loop `will execute`, loop body will execute sum will become, so initially first of course, we sum and count initialize to 0. Now, the first time around the loop execute sum will become 5, count will become 1 and p will become p arrow next, now p is currently pointed to this element and p arrow next is this value which is nothing but a pointed to the second element. So, when this statement executes p will now be pointing to the second element of the list, that is what happen sum become 5, count becomes 1.

Again p is not null, so the next time around sum becomes 12, count becomes 2 and p now points to third element of the list, again p is not null. So, sum becomes 15, count becomes 3 and not p becomes null, because `(())` and p arrow next was null, so now p has become null, and so the loop terminates, and sum divided by count is of course, `5(())` 5 will be returned.

Let us see now how list can be created, so essentially `in` in this example of seeing how we can create a list, whenever we `we` read a data item, we are going to allocate memory chunk large enough to hold struct item and store the data item in that field.

(Refer Slide Time: 10:26)

Creation of Linked List

- When we read a new data item, allocate memory chunk large enough to hold a **struct item**. Store the data item in the value field.
- Library call **malloc** is used for dynamic memory allocation.
- Next "insert" this new item in the exiting list.

Now, for allocating memory of a given file, we are going to use library called a malloc, which is the library call used for dynamic memory allocation. So, once you have allocated the memory for a struct item dynamically, then we initialize the data value in that to the value that we just tried as input. And the list that we already have, we insert this new item in the existing list. So, in the first example, that we are going to see, what we are going to do is at the new data item that we read, will be inserted in the beginning of the list that is already there. So, let us see the code to do that.

(Refer Slide Time: 11:07)

```
#include <stdlib.h>
#include <stdio.h>

struct item *create_list(void) {
    struct item *list = NULL, *p; int value;
    while (scanf("%d", &value) == 1) {
        p = malloc(sizeof(struct item));
        if (p == NULL) {
            printf("Cannot allocate more memory\n");
            exit(1);
        }
        p->value = value;
        p->next = list;
        list = p;
    }
    return list;
}
```

The diagram shows a linked list with four nodes. The first node contains the value 5 and its next pointer points to a second node. The second node contains the value 3 and its next pointer points to a third node. The third node contains the value 5 and its next pointer points to a fourth node. A pointer labeled 'list' points to the first node. A pointer labeled 'p' points to a new node (value 4) that is being inserted at the beginning of the list. The new node's next pointer points to the first node (value 5).

Well first for using the malloc function you need to include the file at the lib dot h, so that has to be included in all programs that use malloc. Now, very few function create list, which will create a list of read a number of data items from the input, and create a linked list of those data items, and what it return is a struct item star, which means I will return the pointer to be first element of `(())` created.

Now, we were the local variable, list is the local variable which is again of types struct item star, so and it is initialize to null, and that we call currently the list is empty. And so therefore, we have made the pointer list point to null and p will point `to the` to the structure that we get we dynamically allocate every time, and value is the value that `(())`. Now, as long as reading an integer results in success can of returns one, what we do if we allocate a chunk of memory using malloc, the argument to malloc is `is` size in byte that `we want to of` the memory generated you want to allocate.

Now, recall that the size of any type can be obtain using the size of operated in p, so size of strut item uses the number of size in number of bytes of the strut item, so that the number of bytes that we done `(())` want to allocate. So, we specify that integer as we assignment to malloc what malloc return is a pointer to this dynamically allocated structure.

But, you for some reason memory allocation phase for example, if we have allocated, so much memory dynamically at the system has run out of memory then malloc returns null. So, it is important that whenever we use malloc, we check that the return values is not null, because if your careless and we use up too much memory malloc might return null, so if p is null we `(())` message and exist.

So, assuming p is not null, we such p value of p to the value that we write here, and these two are in certain this new element in the list, let us see how exactly that is happening. And finally, when this loop terminate that is when we could read any more integer values then we return list, now let us see how exactly all this is really happening. So, we have this variable list and p list is initialize to null p is not initialize, let us the first data value that we read is 1.

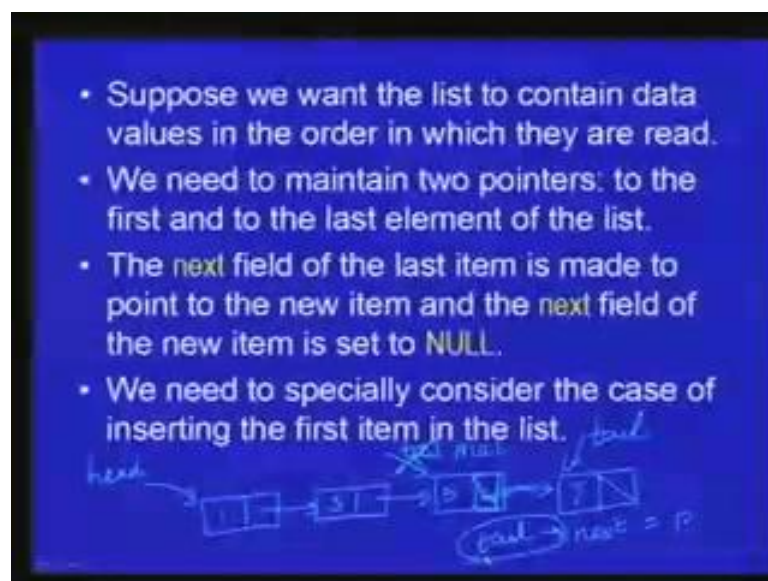
So, this allocates a new structure for this and stores the pointer to this structure in p. So, this is what happens and then we store the value that we write in the value field of this structure; now p arrow next is list, now the value of list write now is null. So, p arrow

next will also become null, so that means, this value the next field of this structure becomes null and the next statement is list assign p which means, at least is no longer null, list is now pointing to this element. Now, let us assume that we read one more integer value, and let us say **we** we happened to read the value 3, so again p is signed malloc a structure of size strut item. So, so the again another structure of the same size are allocated somewhere in the memory, and now p is pointing to there and p arrow value is initialize to 3, p arrow next this time is list.

So, what is being done is such this next field of this new element is made to is being made to point to be first element of the existing element. So, now, this 3 becomes be first element of this list and therefore, the value of list is change to point to this new first element. And **(())** so when the next value is **(())** another structure will be created, and structure 5 is **(())**, and the next will **will** be made to point to the first element of the currently existing list.

And the list pointer will be updated to point to the first element, so that this will become the first element. So, if you at the end of the if you write 1, 3, 5 the list will look like this, please go through this example once again to make sure that you understand exactly, what is happening here, and find the value of listed returned (Refer Slide Time: 15:56).

(Refer Slide Time: 16:14)



Now, for in this course what was happening is that, the list the order of elements in the list is reverse of the order of elements, of the order in which element the values are read.

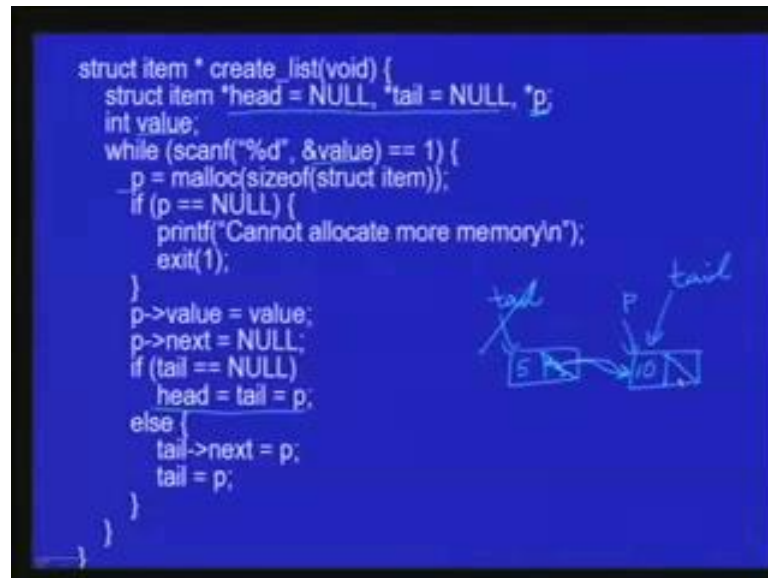
Now, suppose you want to create the list in which the elements are in the same order as the order which ever read, now what we have to do for saving that what will have to make sure is, that if have an existing list of; let us say p items, 1, 3, 5 this is null and we want to insert 7.

So, this is a pointer to the first element in the list, let us `(())` the head of the list and now when we, let us a read 7, now this element must be inserted at the end of the list that means, this pointer value must be updated to point to this and this pointer value must be made null. So, therefore if we want to update this field of this structure we must somehow obtained a pointer to this structure, now one way to do that could be to traverse list once again, and reach to that element, and then search next field of that element to be newly allocated structure.

But, obviously, that it going to be extenders, what is better p is `(())` maintain a pointer to the last element of the list, and when we insert one more element since, we already have the tail pointer all we have to do is set be next field in the structure pointer to volatile pointer to the `new` newly allocated structure. And update `(())` to point to this structure, and that is what it is going to happen in the `(())` just drop that you are just going to see.

But, we have to be careful, because if `in in in` initial is the list is empty, so both had until are going to be null. So, if we try to search tail arrow next to p of where p is the pointer to the newly allocated element, then if the value of tail is null, then we are dereferencing in null pointer, which your result is non predictable behavior, which will probably call program to crush, and so this must not be done. And therefore, we need to specially consider the case, when in certain the first item in the list.

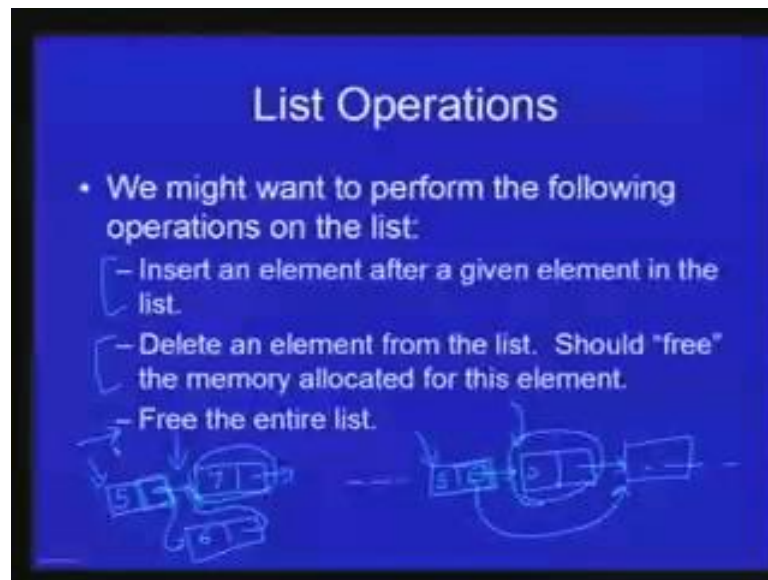
(Refer Slide Time: 18:38)



So, let us look at the code, so we initialize head and tail to null and p will be the pointer to be newly allocated element value is as before, so we read a value and we allocate a new structure for holding this value. So, if p is null we print (()) etcetera, so this p point to the new value, to be new structure that has been allocated, the value field is set to whatever value we read, let us 10, next field is set to null.

And now there are two cases a tail is null; that means, that this is going to be the first element in the list, and so head and tail are both set to p, because this will be first as well as the last element of the list. Now, if tail is not null then tail is pointing to an existing last element of the list, the list has at least one element, let us say this happen to be 5. So, if a tail arrow next to p, which means this tail arrow next, was null initially, so we make it no point to the newly allocated structure and tail is the 2 p. So, that now tail also points here, because this element is now the last element in the list. Let us now considers some other operation for the link list that we might want to perform.

(Refer Slide Time: 20:07)



We might want to insert an element in the list, and we might want to delete an element in the list, now when we delete an element from the list, then we also need to **(())** or de allocate the memory assigned to this particular element in the list. Because if we do not that then what might happen is at the keep allocating more in memory, more in more memory without de allocating any memory and ultimately we might or not of memory.

So, it should good practice to always free or de allocates memory as soon as it is no longer required, and we will see how to do that. Now, when we want to insert or delete an element in the list, you can see that the value of the next field in the previous element is going to change. So, for example, if we have this list **(())** 5, 3 this is, and we want to delete this element from the list, then what will is going to happen is that this value must be changed to this value. So, that this structure is no longer part of the list, so therefore, if we want to modify this value, we must have a pointer to this element of the list.

So, that means, we must have a pointer to the element the before the one that has to be deleted; and similar when we want to insert an element in the list, so let us say this is something like following. Now, suppose we want to insert an element between these two element plus 6, now we allocate a new structure.

Now, what will is happen is that, this value the **the** value of next in the element before **(())** before the position of the inserted element must be changed to point to this, this must be change to point to the next element in the list. Now, if we have a pointer to this then

we can always obtain a pointer to the next element in the list, but if we have a pointer to particular element in the list, we cannot obtain easily the pointer to be previous element in the list. So, therefore, for both insertion and deletion, what is easy to do is to assume that they are given a pointer to the element in the list just before the element which has to be deleted or just before just **just before** replies, we are the element has to be inserted.

And third operation that will consider will be to free the entire list, so we are done with the entire list as a whole, **(())** delete or de allocate all the element of the list, then will have to go through the list and de allocate each structure in the list, each element in the list individually. So, look at course method to do all the three operation, let us say the first operation that is insertion of an element after a given element in the list.

(Refer Slide Time: 23:09)

Insertion in the List

```

struct item * list, (*p, *new;
int v)

/* Insert value v in list after element p */
if ((new = malloc(sizeof(struct item))) == NULL) {
    ...
}
new->value = v;
if (p == NULL) { /* new will be the first element */
    new->next = NULL;
    list = new;
} else {
    new->next = p->next;
    p->next = new;
}

```

So, here **(())** the code might look like, so let us assume that list is the pointed to be first element in the list, and p is the pointer to the element after which the new element has to be inserted, and v is the data value for the new element. So that means, that list is pointed to the first element in the list and p is a pointer to the element in the list after which this element has to be inserted. Let us say can example, may be this is null, and this is list is pointed to be first element in the list, and let us we want to insert the new element here, so p is a pointer to the element just before that.

So, then **(())** we have to of course, allocate in new structure and we do that, and store the pointer to that structure in the variable new, set the value to v the value gets needs to be

inserted. So, let us put the value v here, now let us assume that p is not null, if p is null will assume that what we want to do is to insert the new element as the first element in the list, and that again will have to be specially handled, because we cannot dereference a pointer to the null pointer.

Let us assume for the moment p is not null, so we execute this `p->next`, so what happens is at `p->next` is set `p->next`, now p is pointing to this structure and the value of `p->next` is pointed to this structure. So, new `next` which is this box, the defined a pointer to the `next` in the list after p and finally, `p->next` is assigned new, now this `new` structure p's next element they should be pointing to is a new structure.

So, this is changed pointer new, so essentially now the list looks like 1, followed by 3, followed by v, followed by 5, followed by whatever else which might have been null. Now, the second case is when we want to insert the new element as the first element in the list, and let us assume that in that case the value of p is specified as null. So, if p is null; that means, that we want to insert the new element as the very first element in the list, let us assume that this is an existing list and we want to insert the value 1, let us say, so in this case p is null.

So, `list` type of going to happen is at the value of the variable list itself needs to change, because list remembered by convention will always point to the first element, and the new element will become the first element. And the `next` field of the new element must point to what will become the second element and therefore, `list` what is currently the first element of the list.


So, new `next` is `I am sorry`, new `next` is `NULL`, so the value of new `next` should be that to be current value of list insert will mistake here, this should be list here. And finally, list is said to be, to point to the new element to which new is a pointer, so list is `new`.

(Refer Slide Time: 26:57)

Deletion From List

```
struct item *list, *p, *q;

/* delete the item in the list after the item p */
if (p == NULL) { /* delete the first item in the list */
    q = list;
    list = list->next;
} else {
    q = p->next;
    p->next = q->next;
}
free(q); /* free the memory allocated to the deleted item */
```



So, that inserts the element in the list after the element p, let us now look at deletion from a list and again we assume that list points to the first element in the list, and p is a pointer to the element, the one after which the element after the one pointed to that p has to be deleted. So, again taking an example if we want to delete this element, then will search p to this and if you want to delete the third element will assume that p points to second element (Refer Slide Time: 27:14). So, where assuming that p is pointing to be element the one after which has to be deleted, now again is p is null, and then we are assuming that the first item in the list should be deleted.

So, because (()) points to the first element itself in the second element will get deleted, so if p is null then what should be done is that list should point to the second element of the list to the second element of the existing list. And the first element should be of the first structure should be de allocated memory should be freed. So, now what is your doing is such that using the variable q to point to the element which is being remove from the list, so that we can see the memory being (()) to this element.

So, q is set to list if p is null remember, that if p is null, let us say p is null, so if p is null then very first element of the list has to be deleted, so q is made to point to the list. So, q will always point to the element which is going to be deleted and list (()) list arrow next. So, because the second element in the previous in the existing list not becomes the first element, and list arrow next is a pointer to the second element. So, list is change to point

to the second element, now this element is no longer in the list, but we still have a pointer to list in the variable q. Now, if p is not null that means, you do not want to the list the very first element in the list, so let us assume as an example, that you want to delete the second element in the list.

So, in this case if we want to delete the second element in the list both list and p are pointing to the first element, now what should happen is essentially that this pointer should **(())** structure and point directly to this third element. So, first we set q to point to the element which is to be deleted, and we should always the p arrow next because, element next p has to be deleted, so q is set to that. And p arrow next is set to q arrow next, so p arrow next is this box and the value of such will become q arrow next which is this pointer. So, essentially this pointer **(())** moved and it points directly over here.

So, this element has not been removed from the list, but again we have a pointed to this element in the form of q and finally, once we have moved element from the list. We must allotted free the memory allocated to the deleted item to do that use the library call c, it does not return anything. And the argument is simply a pointer to the chunk of memory which you want to delete, much very careful that the pointer, that you supply must point to a chunk of memory at memory which was dynamically allocated using malloc.

You cannot for example, if you specify the address of some variable **in the** in the memory, in the program then this will have very unpredictable results and also one say chunk of memory has been freed. Now, the pointer few to the chunk of memory must not be dereference, because it is now pointing to a chunk of memory which has been de allocated. So, we will in the next example, see the consequences of how **how** that might arrow new happen, and what might happen in that case.

(Refer Slide Time: 31:04)

Freeing the Entire List

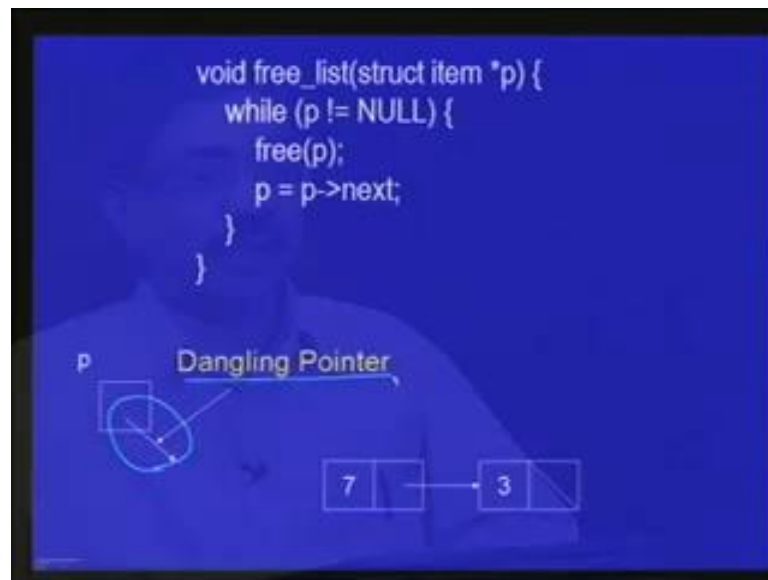
```
void free_list(struct item *p) {  
    while (p != NULL) {  
        → free(p);  
        → p = p->next;  
    }  
}
```

• What is wrong with this code?

So, next what is your going to do is to try, and free the entire list, so we are writing a function free list which takes a pointer to be first element in the list, and frees the entire list that is essentially it is going to traverse the list. And each element in the list is going to be freed. So, memory **memory** assigned to each element in the list is being de allocated.

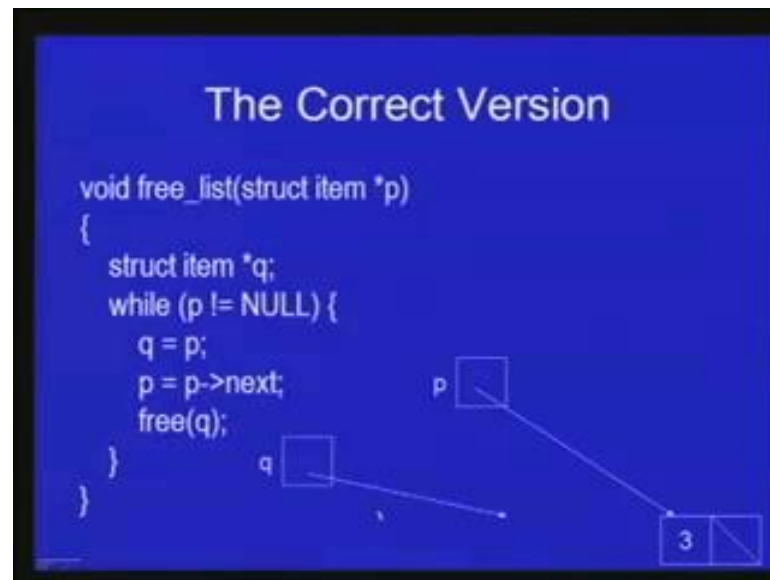
So, this is the first **(())** writing this function and this comment choose there is something wrong with this, so what we do is traverse the list again using the same kind of a value is while p is not null is free p and then a **(())** p to point to be next element in the list. Now, what is wrong with this code you set p is being dereference, where occur which has been freed. So, the chunk of memory has been freed, so it no longer access and now if we try to dereference a pointer pointing to this remember that, p arrow next is exactly the same as star p dot next.

(Refer Slide Time: 32:20)



So, the dereferencing p which is pointing to **(())** use of memory which has been de allocated and that will unpredictable results, let us see that more **(())** example let us assume that p is pointing to the first element in this list 5, 7 and 3, now p is not null. So, the first time around the list execute, free p will essentially de allocate this structure, so which a pointer **(())** as argument, so this structure p **(())** to exist in function. Now, you can see that p is pointing to a structure which no longer exist and such a pointer is called dangling pointer, and now when we try to find p arrow next, that will result in unpredictable behavior; that is we call the value of p is what is known as the dangling pointer? It is not pointing to a valid region in the memory.

(Refer Slide Time: 33:08)

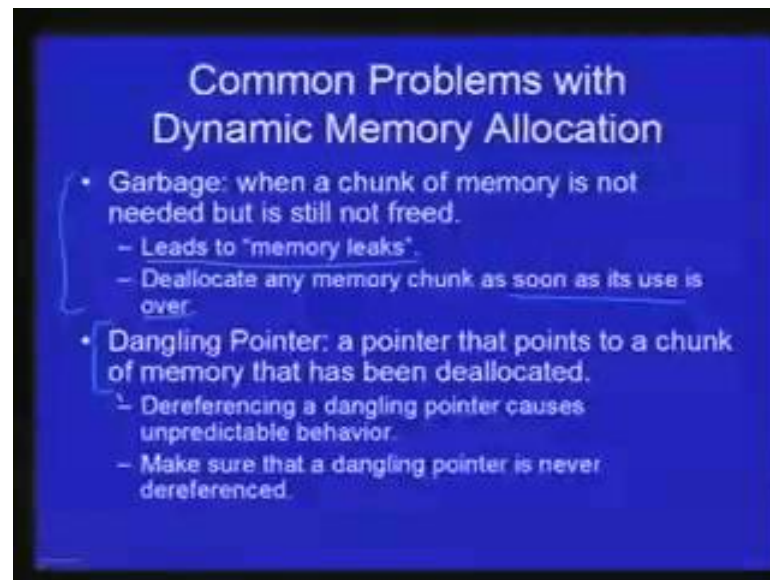


So, this is the correct version, what we have to make sure essentially is that we do not dereference the pointer once we have freed the memory pointed to `(())` at pointer. So, what is we are doing is that in `(())` `p` as a `p` arrow next before doing the free, but if we do `p` assign `p` arrow next then we already loss the whole value of `p`.

So, what we do is store the old value of `p` in `q` and then you `p` `q` in strut, so that is specially seen it before that dereferencing of dangling pointer is not happening, let us see how it is being to execute. So, let us again assume that a `p` point to be first element in this list and now `p` is not null, so the loop body execute `q` is assign `p` then `p` is assign `p` arrow next, so `p` points to the second element, now free `q` happen, so the first element get de allocated.

Now, `p` is still not null, so we go back to the loop condition `p` is still not null, so the loop body will execute once more `q` is again assign `p`, so `q` now points to second element `p` is assign `p` arrow next and then free `q`. So, the second element get de allocated, then `p` is still not null, so will execute the loop one more time `q` will assign `p`, `p` is assign `p` arrow next `p` arrow next in this case is null, so `p` becomes null. And free `q` results this being last structure being de allocated, and so the entire list has been freed.

(Refer Slide Time: 34:44)



Now, to summaries there are two common problems with dynamic memory allocation, and you will find that unless your careful, you will easily make this mistake and one has to be extra careful again these kind of mistakes. The first kind of problem result, when we do not free a dynamically allocated chunk of memory even when we are finished using this, so such a chunk of memory is called garbage appropriately, because it is still in the, it is **it is** still not read, so it cannot be reused, but is not useful **(())**

So, this leads to what are known as memory leak essentially what happens is that programs keeps allocating memory by we does not free memory, then ultimately it might lot of memory, and that is like **you know** having the memory leads the bit. And the solution to this problem; of course, must be careful and de allocates any memory chunk as soon as its use is over.

And the second problem that is **(())** just use the problem of dangling pointer that is the pointer that points to a chunk of memory that has been de allocated. And as for dereferencing a dangling pointer causes unpredictable behavior, and the only way to avoid this is make sure that a dangling pointer is never dereference **(())** must always be careful not to that.

So, that is the end of today's lecture and as the method **(())** this was the last lecture in the course. So that is end of the course as well, and I hope that you found the course **course** useful as well as interesting. But before I teach your leave I would like to remained, you

that programming is not something that you can learn the reading a book or by watching videos, the only way you can learn to program is to actually write programs. And now just on paper, but also try out change actually on a computer you will **(())** make mistakes and its course mistakes that of course, at one launch from. So, **so** write lots of programs have fun doing it, and thank you for watching this series.