Programming and Data Structures
Prof. N. S. Narayanaswamy
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 09
Stack ADT and an Applications

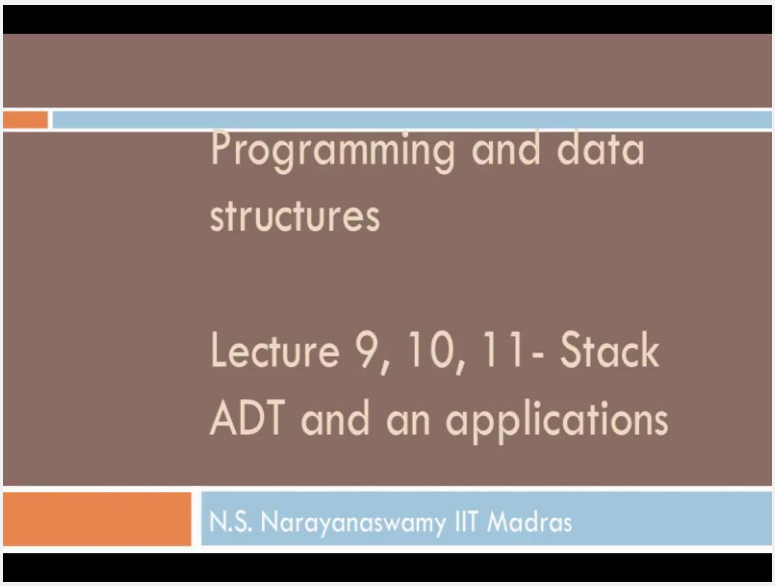
Welcome to this week of lectures on stack, the abstract data type. And recall that we have over the last three weeks looked at different data structures. So, we looked at the link list data structure, doubly link list data structure, then we looked at the array, we compare the three of them for their efficiency for different implementation purposes. And one of the most important tools that we repeatedly use was the power of recursion. So, if you look at all the programs that we have written so far, there was definitely at least one recursive function which was implemented.

You will also be I mean reasonably recurred, because of the programming exercise that we seen so far which have use recursion, and we will have definitely about three or four more interesting programming exercises, just after this set of lectures are over. In today's lecture or in the lectures of this week we are going to have three short lectures, each one of approximately 20 to 25 minutes long.

The first one will be an over view of the stack data type, it is applications and the different kinds of purposes the stack data types serves. After that, that will be a power point presentation and just after that we will have two 20 minutes sessions on very important applications of the stack data types. So, this might be little more than 20 between 20 to 25.

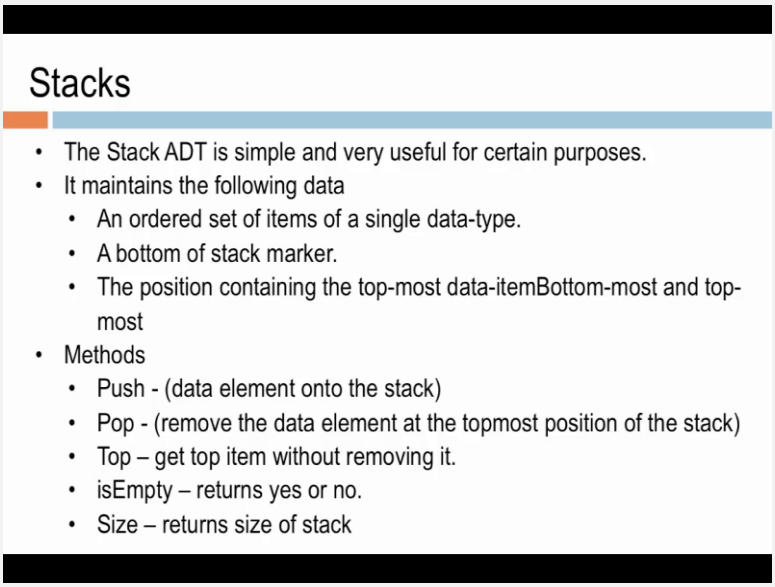
So, let us start off with the description of stack data type and the understanding of what kind of methods are there in the stack data type and what are the different data items and also an understanding of the different problems which use the stack data structure in a very fundamental way. So, let us just go to the slides.

(Refer Slide Time: 02:11)



So, this is lecture numbers 9, actually it is lecture number 9, 10 and 11, because we most likely not come back to the power points slides, we just go directly to the different programs that we have written associated with the stack data type.

(Refer Slide Time: 02:43)

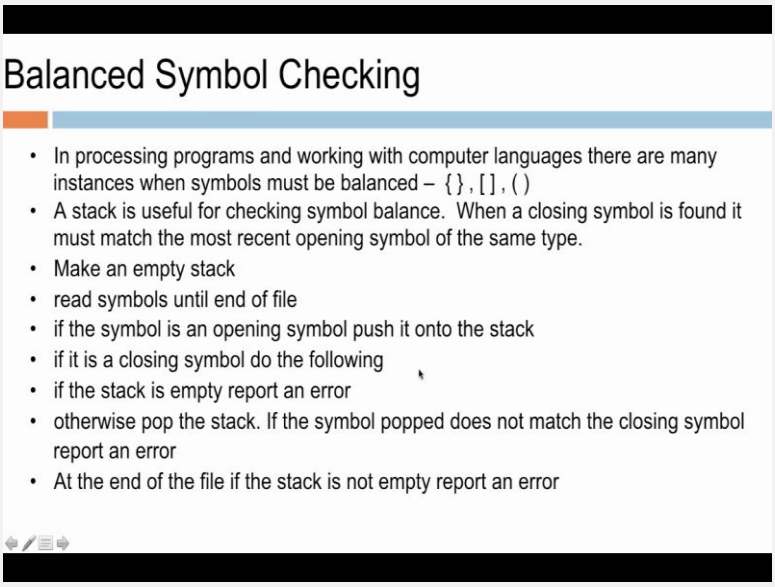


You will see and I am sure those of you have already program will know the stack is a very, very simple data type and it is very useful for many purposes, as we reach the tail of this lecture we will list of lots of applications of the stack data type. So, if you look at the stack of abstract data type, you have the following data items, you have ordered a set

of items of a single data type. And there is also one data item which corresponds for the, what is called the bottom of the stack marker. So, the value of this variable will tell you whether the stack is empty or not.

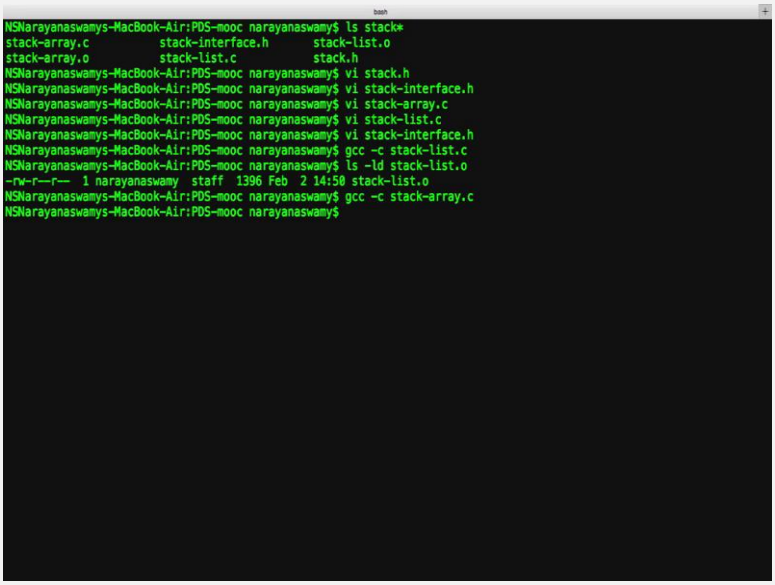
And then we have the position of the top most data item which can be accessed in query. A methods that we have are functions which taken a value and the name of a stack and push the data items onto the stack. We will see what push actually does and then there are methods, there is a method called the top method which removes a single element from the top most position of the stack. And top also allows you to look at the top most element without removing it, then we have the standard membership queries which are the isEmpty function which returns a Boolean value, we also have a function which returns the size of the stack.

(Refer Slide Time: 04:33)



So, before we go to a set of applications let us now go to an implementation of these methods, just for you to understand what is a return value and what are the arguments to each of these methods. So, for this let us go to the set of programs that I have written.

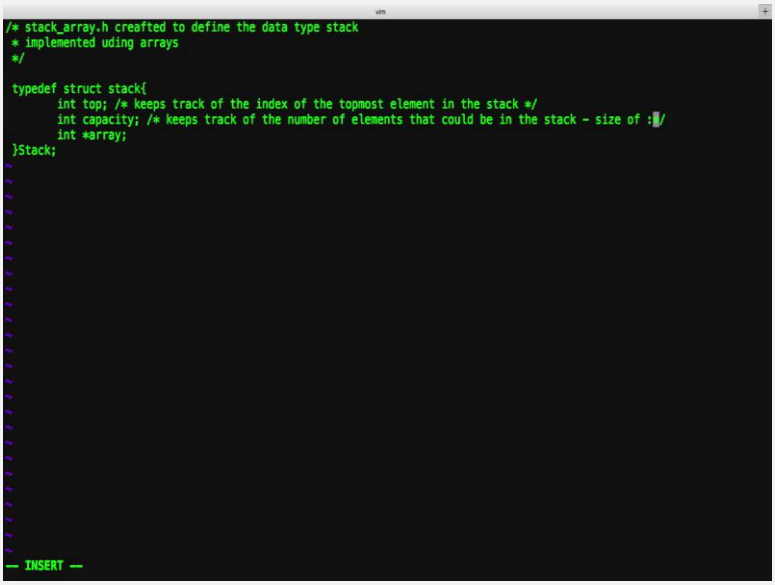
(Refer Slide Time: 05:03)



So, here are the functions, stack dot h as usual is a definition of the data type, one node of a stack, the stack interface dot h is a function that will be used by the stack interface dot h is a header file which will be used by a programmers, who want to use this stack implementation. There are two dot c files you will see, one is called the stack listed dot c and other is called the stack array dot c, these are the array and the list implementations of a stack that is, the stack is implemented by storing the elements in a list and then we have implemented all the methods that was listed in the slides.

Similarly, here the stack is implemented, the elements in the stack are arranged inside in an array and the methods are implemented assuming that the stack is actually arranged in an array. It is possible to write a single function, but it is better to actually break it down and write it in this particular fashion. So, let us just look quickly at this piece of code and that will essentially form this particular lecture module.

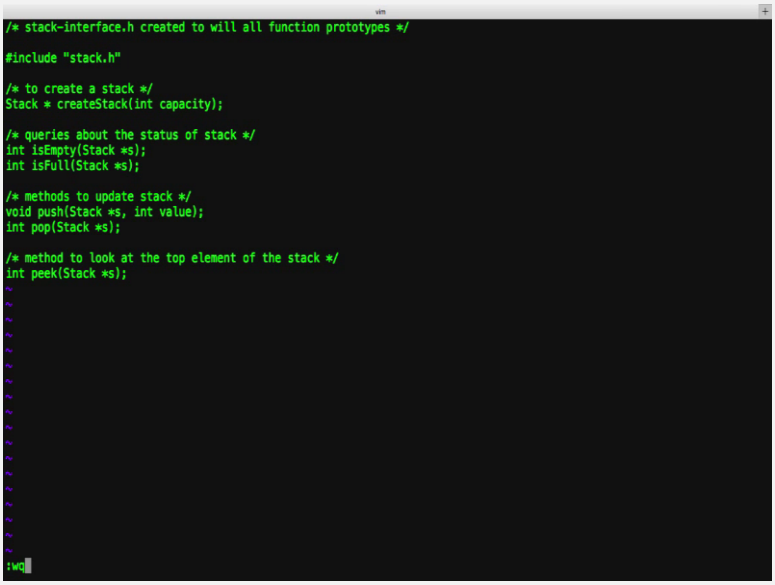
(Refer Slide Time: 06:29)



So, the stack dot h is very simple, we are creating a new data type, the type name is stack, the type is actually struct stack and observe that there is one variable called top which keeps track of the index of the topmost element in the stack. There is an element called capacity which keeps track of the total number of elements that could be stored in the stack. So, this is bit technically incorrect, keeps the number of elements that could be stored in the stack that is the size of the stack.

Otherwise, the size of the stack, the third data item is the stack itself, observe that this is stored. This stack actually stores an array of integers and the size in the array of integers will be given by the capacity. So, we will see how to initialize the stack and how to say how large the size of the stack is and so on and so forth by looking at the function implementations (Refer Time: 07:47)). Now, let us look at the stack interface which is what programmers use.

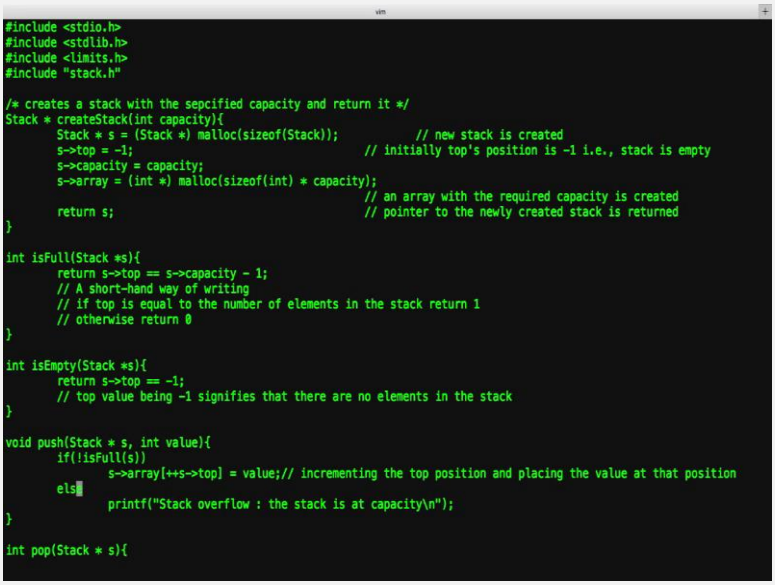
(Refer Slide Time: 07:53)



So, first of all there is a function called create stack which takes an argument capacity and returns a pointer to stack. The other two functions are isEmpty and isFull, both of them return Boolean values and the last two methods are methods to update the stack which is just to push a value onto a stack, which is pointer to by the variable s and this one is to push, is to pop a values in this stack.

What is to pop a value in the stack? You take an element from the top of the stack and here is a peek function which takes a pointer to the stack and looks at the top elements, in other it is different from pop, pop returns the top, pop removes, pop looks at the top element in the stack and removes it from the stack, whereas peek only just looks at it and does not remove it from the stack. So, these are the interface files that we will repeatedly use ((Refer Time: 09:05)). So, let us go to the implementation files, so that you have a clear understanding of...

(Refer Slide Time: 09:12)



So, let us just look at the first function which is creates stack, it takes as an argument the integer capacity and you can see that the new stack is created here using the malloc function and the top most element in the stack is set as minus 1. When it is set as minus 1, it is assume that this is a convention or an indicator that the stack is empty. And the value of the capacity of the stack is kept as the argument. The capacity is stored in the variable called capacity associated with the stack.

And this one ensures that the stack, this one ensures that we create an array of a certain size and let it point to the stacks. So, that is something very interesting going on here, so first of all we create a stack object, it is called s and then we initialize the top of stack value to minus 1 indicating that the stack is set to empty, we seen the capacity of the stack is the given value capacity and the last one, we make the s point the array feel point to an array of integers, how many integers, capacity number of integers.

So, this is... So, here we created an array with the required capacity and now we return a pointer to the stack s. So, therefore we have created one stack here by this procedure. Now, let us look at the simple functions, so when would be the stack be full? So, the stack would be full, if the index of the top most element is the capacity minus 1. So, let us just see this, top will essentially store the index of the top most position in the array, remember that the array is the index from 0 to capacity minus 1.

Therefore, if the value of the top most element is capacity minus 1, then it means that you have to return 1 saying that the stack is full; otherwise, you return a 0. So, observe here that it is a short hand way and this is very interesting, this short hand way of returning a comparison. So, many of you might have seen such expressions return only inside if and while statements, but observe that return this, the result of this comparison can be returned.

So, what is this comparison just to read write**,** it checks if the index of the top most element is capacity minus 1. If it is indeed capacity minus 1, then it means that stack is full. When would the stack be empty? We have already had this convention that the stack is empty, when the top most element is pointing to minus 1 has the value minus 1. So, therefore here is this comparison, you check if the topmost element is equal to minus 1 in the return expression. So, now comes... So, these are the decision functions associated with the stack and here is a stack constructor function or the creator function.

(Refer Slide Time: 13:16)

```
        return s;                                       // pointer to the newly created stack is returned
}

int isFull(Stack *s){
        return s->top == s->capacity - 1;
        // A short-hand way of writing
        // if top is equal to the number of elements in the stack return 1
        // otherwise return 0
}

int isEmpty(Stack *s){
        return s->top == -1;
        // top value being -1 signifies that there are no elements in the stack
}

void push(Stack * s, int value){
        if(!isFull(s))
                s->array[++s->top] = value;// incrementing the top position and placing the value at that position
        else
                printf("Stack overflow : the stack is at capacity\n");
}

int pop(Stack * s){
        if(!isEmpty(s))
                return s->array[s->top--];// returning the value at top's position and decrementing the top position
        printf("Stack underflow : there are no elements in stack\n");
        return INT_MIN;                 // some sentinel value is returned to signify failure
}

int peek(Stack * s)
        if(!isEmpty(s))
                return s->array[s->top];                // note the difference between pop and peek
        printf("Stack underflow : there are no elements in stack\n");
        return INT_MIN;                         // some sentinal value is returned to signify failure

"stack-array.c" 47L, 1576C written
```

Now, let us look at how to push a value on to the stack. So, when should you push, so let us just look at the logic, so the stack object a pointer to the stack is created, this pass to the push function, it does not return any value and it also takes the argument value here. So; obviously, the first check is to check that… So, the first check is to check that the stack is full or not. So, the condition is if the stack is not full, then we look at s pointer at array, incrementing index first and then place the value in the position, so it is a very

simple.

So, before you put the value inside, you incremented and put the value inside. So, that is probably the stack is not full, which means at least one location is free. If it is not full, then it means there at least one cell in the stack is free and you increment the value of the index top first that is what this says. It says plus plus which one s pointer is top. So, before the value is used, the plus plus operator will incremented and the variable value, then the value will be stored inside that particular location.

On the other hand, if the stack is full there is an error message which says that there is a stack overflow, saying that stack is a capacity. Stack has reached it is full capacity and the insert has failed. So, when you normally programmed you can return a failure signal here, here also you can actually return an integer value, but just because I am showing you a simple programming example, I kept the return value to be void and we print out a message.

Similarly, let us look at the pop function which takes in just a stack as an argument and it checks if the stack is empty, if the stack is not empty, then it observe here that you return an element, return the top element first and then decrement the top element by 1. On the other hand it is empty, you say the stack in underflow, there are no elements in the stack and here we keep a constant called int min says that it is an internal value which is used to signify failure. You can set int min minus 1 or whatever you want, just to say that the insertion has failed.

So, comes the third function, so observe push pushes the value onto the top of the stack and increments the top value by 1, pop gives you the top most value in the stack and then reduces the index by 1 and what is peek do, peek just checks the top most element. If the stack is not empty, then it just returns the value of the top most element, but does not eliminate it from the stack and we will see how, why this is very useful in a subsequent lecture.

Of course, if the stack is empty it returns the value, it returns an underflow, saying that the stack is empty and the value there can be, it is not possible to read a value from here. So, this is the array implementation of stack, let us just run through this once, there is the most important thing is the constructive function, this creates a stack and then it populates a stack. With these values, initially it says that the stack is empty by setting the

top value to minus 1, it says that the stack has a certain capacity indicating the maximum number of elements that can be stored in the stack and then you associated an array of capacity number of integer elements to be stored and you create this array done and get it.

Now, this creates a stack, then you have the isFull function and isEmpty function which returns Boolean values, then there is a push function which pushes the value into the given stack s. Pop returns the top most element and eliminates it from the top of the stack, peek returns the top most elements and keeps it over top of the stack, peek is like ((Refer Time: 17:49)) to the stack. So, this finishes the study of the array implementation of the stack methods ((Refer Time: 18:00)). Let us look at the list implementation of the stack methods.

(Refer Slide Time: 18:03)

```
/* this is to show how to implement stacks using lists */
#include <limits.h>
#include <stdlib.h>

typedef struct node{
        int value;
        struct node * next;
}Stack;

/* Note that unlike arrays we do not require capacity while using lists
 * so there is no need for createStack() method. It is just creating a node.
 * Since there is no capacity, there is no need for isFull() method.
 */

int isEmpty(Stack *s){
        return s == NULL;
}

void push(Stack *s, int value){
        Stack *newNode = (Stack *) malloc(sizeof(Stack));
        newNode->value = value;
        if(s == NULL)
                newNode->next = NULL;
        else
                newNode->next = s;              // note that every new element is added in the front of the list
        s = newNode;
}

int pop(Stack *s){
        Stack * temp = s;
        int popValue = INT_MIN;
        if(!isEmpty(s))
        {
                s = s->next;
```

Observe that there is a slight difference, here we have a node, the node stores a value and also has a pointer to another node and this, the name of the type is called a stack. Now, therefore whenever you want to insert into the stack, you get a new location onto the stack and whenever you want to delete from the stack, you release a certain element of the stack that is the planned approach here, as suppose of what we did in an array. Recall that in an array we actually created the stack object and associated an array with the stack object.

So, here the stack is composed of a collection of nodes, inside each node is one integer element and are pointer to another node of the same type, in the type name is called stack. So, let us quickly look at the simplest of methods, when is the stack empty when s is null therefore, if s is null then uses return that s stack is empty. How to you push onto the stack? So, what you do is to push onto the stack you create the new node, you associate the value into the value filed of new node and if the stack was initially null, the new node becomes a only elements in the stack and the next pointer points to null.

Otherwise, new node pointer dot next points to s which assuming that s is the top of the stack. And now you return, now you set s to be pointing to new node, so observe that we have augmented or incremented the size of the stack by one unit. So, you create a node, you keep the value in the node and then you check if the stack was empty, in which case new node becomes the first element of the stack.

Otherwise, we set new node to point to the current first element of the stack which is s. And finally, you say s is equal to new node, observe that the advantage here is that you have no limitation. So, the total number of elements in the stack can stored.

(Refer Slide Time: 20:35)

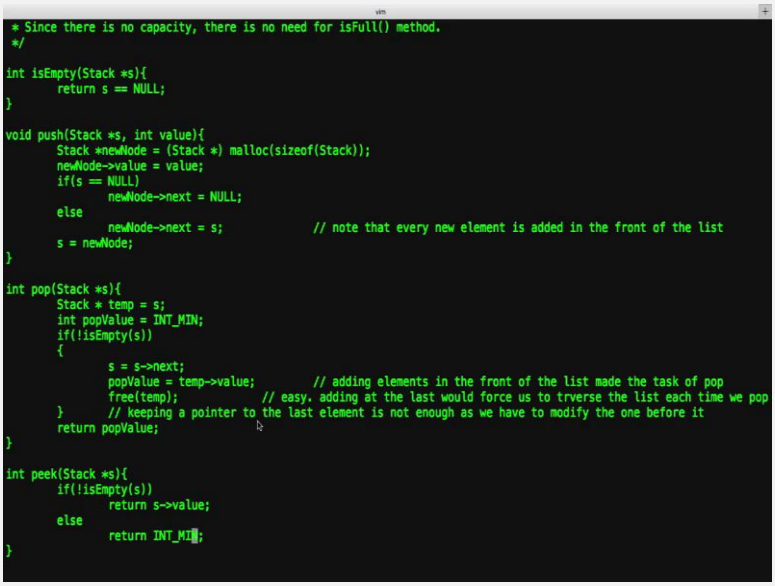
```
/* Note that unlike arrays we do not require capacity while using lists
 * so there is no need for createStack() method. It is just creating a node.
 * Since there is no capacity, there is no need for isFull() method.
 */

int isEmpty(Stack *s){
        return s == NULL;
}

void push(Stack *s, int value){
        Stack *newNode = (Stack *) malloc(sizeof(Stack));
        newNode->value = value;
        if(s == NULL)
                newNode->next = NULL;
        else
                newNode->next = s;              // note that every new element is added in the front of the list
        s = newNode;
}

int pop(Stack *s){
        Stack * temp = s;
        int popValue = INT_MIN;
        if(!isEmpty(s))
        {
                s = s->next;
                popValue = temp->value;         // adding elements in the front of the list made the task of pop
                free(temp);             // easy. adding at the last would force us to trverse the list each time we pop
        }       // keeping a pointer to the last element is not enough as we have to modify the one before it
        return popValue;
}

int peek(Stack *s){
        if(!isEmpty(s))
                return s->value;
```

Now, let us this look at the pop methods in the stack, so that the pop will take away the head of the stack. So, the top most element of the stack, so for that you check if stack is not empty, first if the stack is not empty then you say s pointed are next. But, before that you stored the value of x in a temporary variable or temp, then the value to be return is

called top value, top value is initialize is some global constant,. it is the top value gets a value inside, the current top of the stack.

Now, here is a very important step what we do is, we free temp and then you return the value which is a top value. So, observe what we would done we make a copy of the top most element that is 10 of the stack, we check that s is not empty, if s is not empty then s is made to the point to the next element in the stack. Then we take the value from temp value that is the value to be return, now you are realise temp of free temp and you return the value that you need a to top from the pop for the stack.

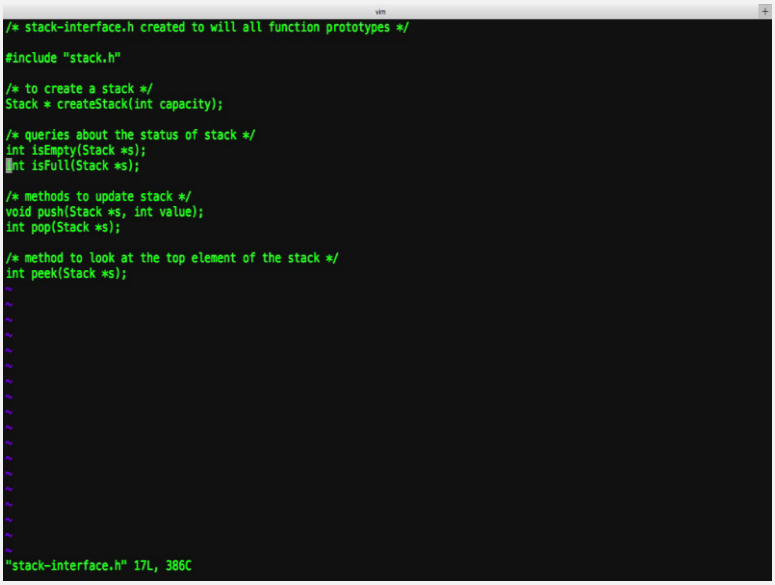
(Refer Slide Time: 22:06)



Here is the peek function which is also very simple, if the stack is not empty then you return the top most element, the value of the top most node in the stack. Otherwise, you return a failure symbol which says that reading from the stack has failed. So, that brings to an end the implementation of the methods associated with the stack abstract data type. So, before we end this short presentation ((Refer Time: 22:46)).

Let us this look again at the stack file that we have, we saw the stack dot h this was very important for the array implementation of the stack. Then we looked at the list implementation was stack and the array implementation was stack and we also look at the stack interface file, which is what a programmer will use and let us just look at the stack interface file.

(Refer Slide Time: 23:15)



Observe here that the stack interface file is independent of whether you are doing an array implementation or a list implementation of the stack, this is the very nice think right. So, there are two different implementations, one is an array implementation, one is a list implementation, the methods which are implemented in the array implementation or the list implementation, the methods no are use the fact that the underline data structure containing the stack values is an array or a list as a case may be.

But, the interface files which is, what is accessible to a programmer do not review the any information about the underline implementation of the stack methods or the stack access methods ((Refer Time: 23:59)). So, this is the very important feature in today's lecture, so not only are we learning about the stack data type. But, we are also learning about certain very professional programming skills that could be very useful are anyone's carrier.

So, we also compiled as you can see two dot o files, let me show the compile commands for you gcc minus c stack list dot c this compilation finishes and the stack list dot o has been created. So, similarly we compile the gcc array dot c, so therefore, two modules or two object files had been created. So, that brings to an end today's lecture just on the stack abstract data type, this is a lecture 1 on the stack.

When we meet next we will look at the different applications of the stack data type and then we will go to different programs. So, whenever we discuss certain application we

will go to the some program and run that program, so that is the plan. So, this brings to an end the first lecture on the stack abstract data type.

Thank you.