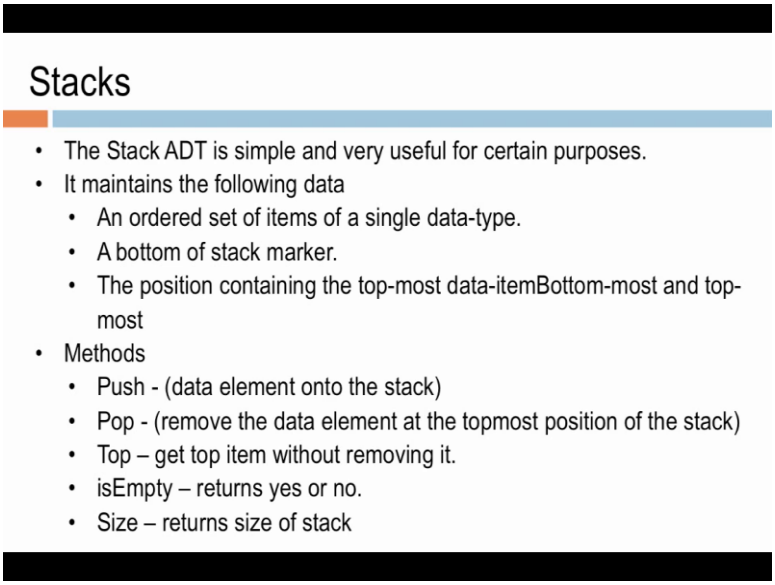
Programming and Data Structures
Prof. N. S. Narayanaswamy
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture – 10
Checking of Balanced Parenthesis

Hi, welcome to the second lecture on the stack data type. In the previous lecture, which was a short 25 minute lecture, we actually looked at the stack abstract data type and the different methods associated with it. We saw two implementations: the list implementation, and the array implementation. When I mean the list… What I mean by the list implementation and the array implementation is that, the stack data are organized in a list or they are organized in an array. And we also saw some interesting programming practices. So, we noticed that the header file really did not expose the information to the programmer as to whether the underlying implementation is the array implementation or the list implementation. So, this is a very important programming approach that you should definitely try and use in your different programming projects.

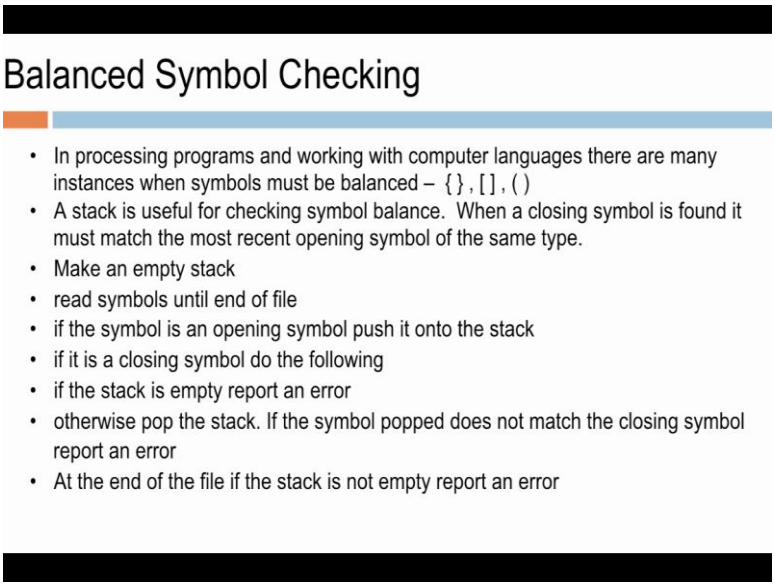
In today's lecture, what we are going to do is we are going to look at one application of the stack data type – one or two applications of the stack data type; and we will go through the problem statement and then go to the piece of code, where this application has been implemented. We look at the… We will look at the function calls made in that particular program; we will run the program; understand what the programming does and stop. That will essentially be another short 20 minute module – 20 to 25 minute module for the lectures of this week on the stack data type.

(Refer Slide Time: 01:44)



So, we already saw the stack abstract data type and its implementation.

(Refer Slide Time: 01:49)



Let us look at the following very simple exercise. So, many of us repeatedly write multiple arithmetic expressions. So, when we write a program, we write many arithmetic expressions that involve different kinds of parenthesis. For example, if you are

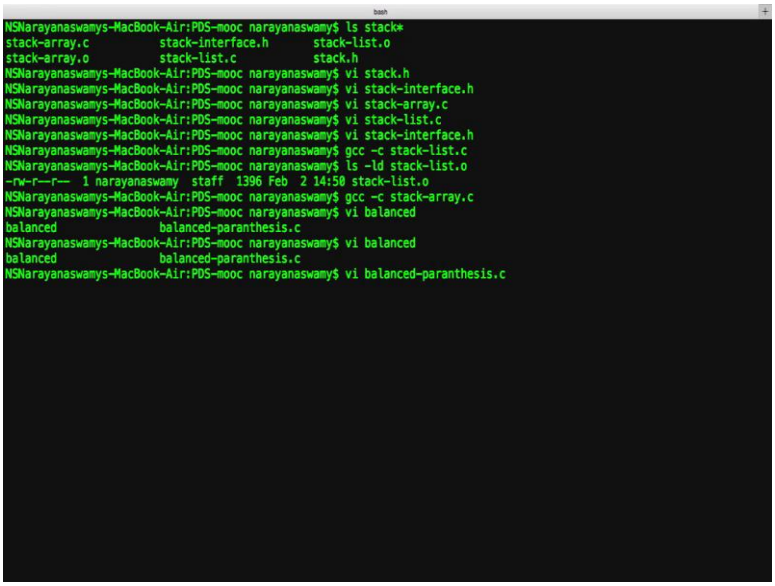
programming in C, this is the standard delimiter. So, for every appropriate function call, then you have an open braces and a close braces. Here whenever you define an array, we use the square brackets; and when you define a function for example, you use these parentheses. So, very simple programming exercise is to check whether in a given expression or in a given sentence, whether the collection of parenthesis alone forms the balanced set. So, what do I mean by balanced set? So, it means that, the open brackets must be followed by the closed brackets. So, if I take a sequence of brackets, I say that, the sequence of brackets is a balanced sequence if every open bracket is followed by a closed bracket – a unique closed bracket. In other words, for every open bracket, there must be a corresponding closed bracket. I mean it cannot be the situation that you have like 50 open brackets and 1 closed bracket; following it? That is definitely not balanced. So, therefore, for every open bracket, you must be able to identify a unique corresponding closed bracket. This is the program.

And the focus of today's lecture is to actually try and understand how to write a program that achieves. So, let us look at the features of the program. So, first of all, most importantly, a stack is very useful in checking symbol balance. And the idea is a following. Whenever we find open brackets, we keep pushing the open brackets on to the stack. Whenever a closed symbol is found, we must match it with the most recent opening symbol of the same type; that is, if you found a flower bracket, you push it on to a stack. Just following it, if you see an open bracket – closed flower bracket; then that closed flower bracket is associated with this particular open bracket. Therefore, when a closing symbol is found, it must match the most recent opening symbol of the same type.

So, how are we going to do this? First, we make an empty stack; then we read the symbols one after the other. Whenever we read a symbol, if it is an opening symbol, we push it on to the stack. Now, if it is a closing symbol and if the stack is empty, then we report an error; otherwise, we pop the stack; and if the symbol pop does not match the closing symbol, we report an error. At the end of the file, if the stack is not empty, we report an error. In this case, it would mean that, there are more number of opening symbols, than closing symbols. So, I hope the logic is clear. The logic is that, you will scan the input sequence of the parenthesis, which can be of these three different types.

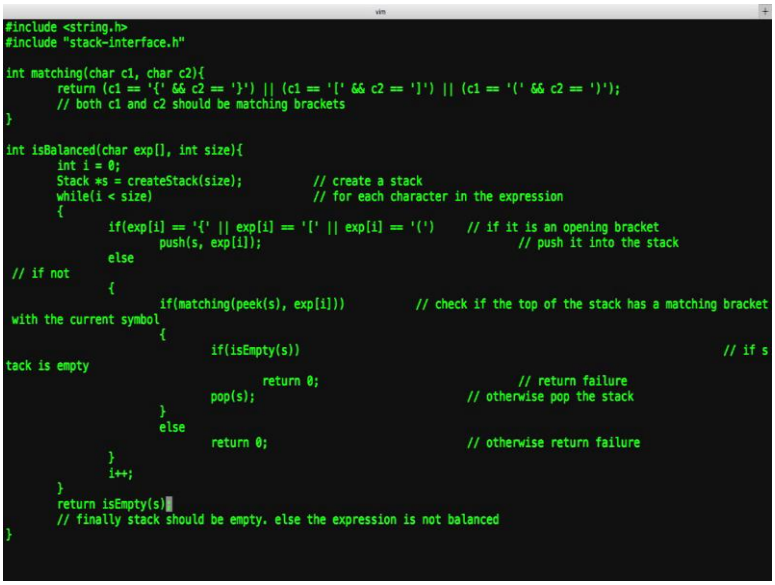
Whenever we find an opening symbol, which could be either this flower bracket or this square bracket or this parenthesis; we push it on the stack. Whenever we see a closing symbol, we pop of the topmost value of the stack and compare whether the closing symbol and the opening symbol just obtained from the top of the stack are of the same type. If they are not of the same type, you give an error message; if they are of the same type, then you are done; you can continue to go through the recursion. On the other hand, if finally, the stack does not become empty, then you report an error. Or, when you are popping from the stack, if you find the stack is empty, you also report an error. So, this is the idea of balanced symbol checking. And as you can see, it is a very simple programming exercise; but it is a very very important programming exercise; all compilers use this extensively.

(Refer Slide Time: 06:19)



So, now let us go back, go and look at our program for this particular task.

(Refer Slide Time: 06:29)



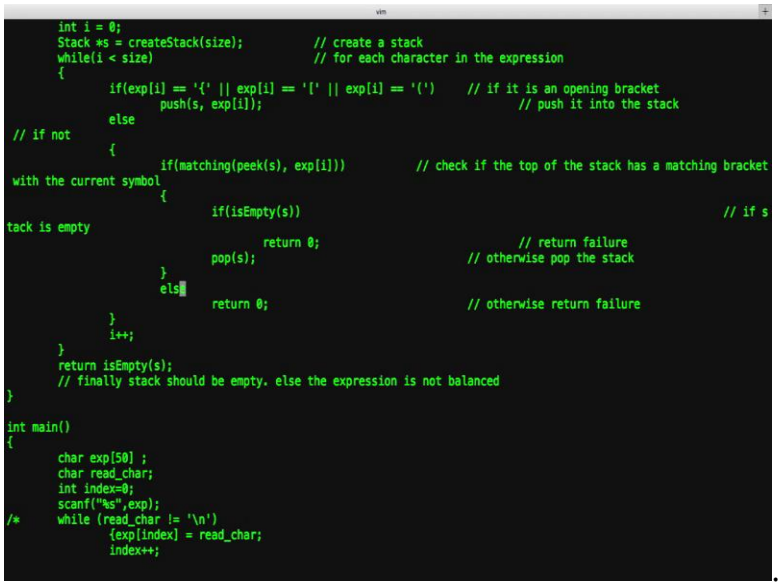
So, let me just quickly reorganize the commands. So, let us just go through this. So, the most important thing is we have a method, which takes two characters and checks if the two characters are of the same opening and closing pair; that is, you return a true if both c 1 and c 2 are of the same parenthesis type; that is, c 1 is an open bracket; c 2 must be a closed bracket; then you return true. Or, c 1 is a square bracket – opening square bracket and c 2 is a closing square bracket; then you return true. And c 1 is an opening parenthesis and c 2 is a closing parenthesis; then you return true. Now, let us directly go to IsBalanced function. And let us look at the arguments for IsBalanced function. This is the crucial function; it checks if the given sequence of parenthesis are balanced or not. The parentheses are in this character array; they are actually a string. And the total number of symbols, which are there, which are of the parenthesis nature – parenthesis type is given by the integer variable size. So, i is a counter here.

So, now, first we create the stack. Observe that, we do not have to worry about the implementation. Now, we know that, s is a stack; s is a stack of a certain size. And now, we look at the given expression and use the stack to check if it is balanced or not. So, i is a counter; it starts from 0 until the last element of the stack. Now, here we say that, if the i-th symbol, which is read from the sequence of the parenthesis is an open bracket; then you push it on to the stack; that is, you push on to s, exp of i – that is the i-th element;

otherwise... So, which we come here; we look at the top of the stack. This is peek; we are not going to pop of the top of the stack; we are going to only peek at the top of the stack and check whether it matches with exp of i. So, therefore, if this matches, then we go here. So, if the stack is empty, then we say that, we will return failure; otherwise, we will pop from the top of the stack.

So, let us understand this. So, you match the contents of the top of the stack with the symbol that you are reading. And if there is a match, you check if the stack is empty. And if the stack is empty, then you return a 0; otherwise, you pop the value from the top of the stack. There is an otherwise here; if they do not match; then you return failure saying that, the given sequence of parentheses is not balanced. Then you go on to the next symbol. At the end, if the control reaches this point, now you have to check if the stack is empty or not; if the stack is empty, then you should return s – the sequence of parentheses, which have been given is balanced; otherwise, you return no.

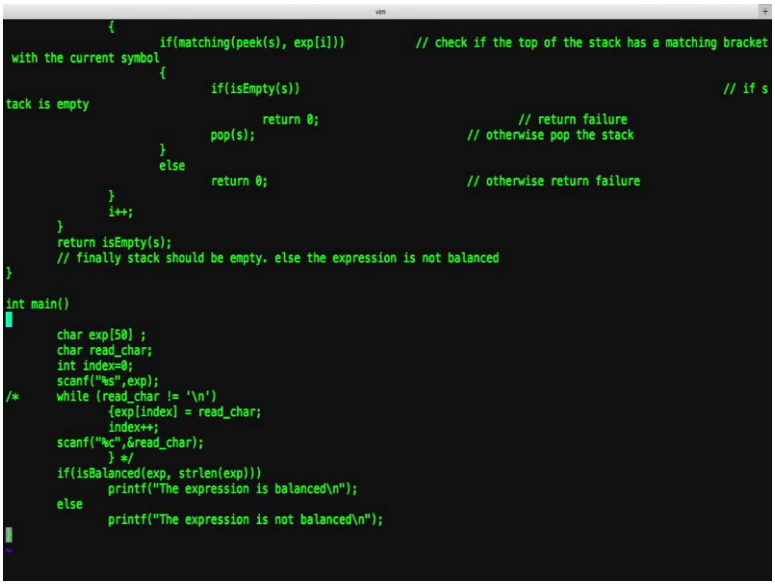
(Refer Slide Time: 11:14)



So, the… Observe that, the implementation exactly has implemented what we had put on the slides. And observe that, it is using methods, which are implemented in the methods to which we have the access to which is obtained from the stack-interface dot h. And really we do not know whether the implementation is using an array or is it using a list

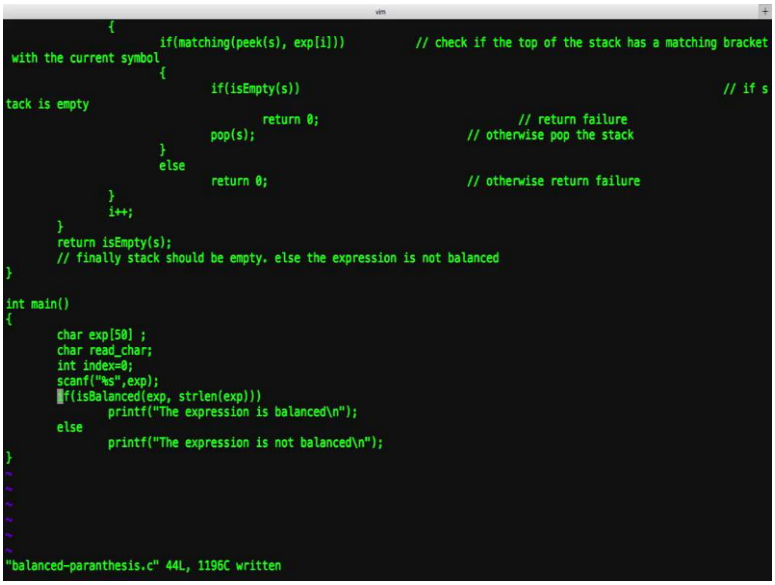
implementation.

(Refer Slide Time: 11:46)



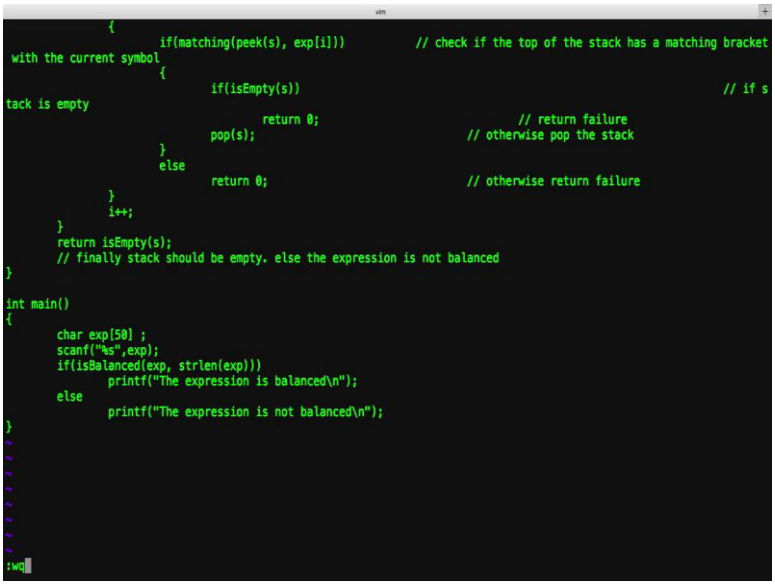
So, let us look at the main function. So, the given expression consists of 50 symbols. So, now, we read this whole string into expression. Observe that, there is no ampersand here; you can actually remove these.

(Refer Slide Time: 12:15)



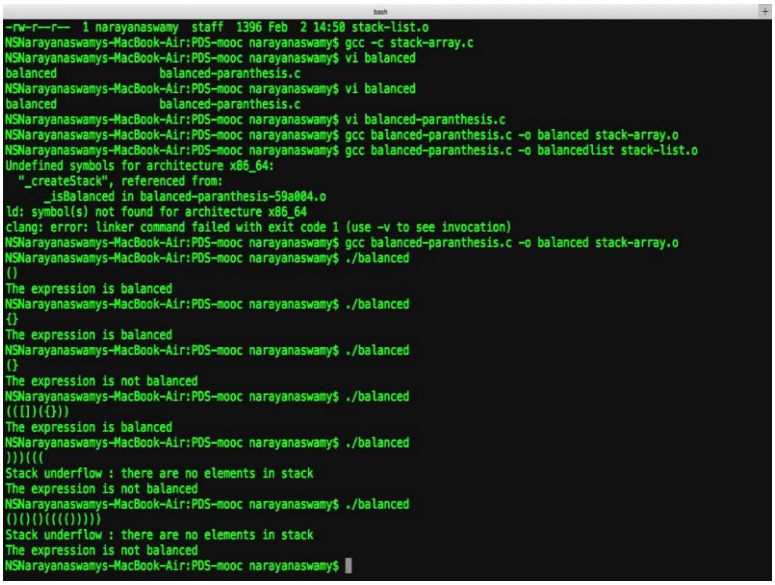
The code is extremely simple and crisp. So, now, you check; if it is balanced, then you print whether it is balanced or not; otherwise, you say that the expression is not balanced. Observe that, we are using a function from the string library, which tells us how long the string in exp is. So, this completes the implementation of the balanced-parenthesis dot c.

(Refer Slide Time: 12:44)



And let us compile it with the array.

(Refer Slide Time: 12:45)



So, gcc balanced-parenthesis dot c. And let us output it to balanced. And let us compile it with stack-array dot o; that compile. Now, this is using the list implementation and let us use the… So, that was an error. So, I have forgotten something. So, I take this away – take this back. So, therefore, what we did just now was to compile it using the array implementation of the stack methods. So, let me just compile this again. Now, let us run the balanced parenthesis checker that we have just written. So, let us give the simplest of exercises; it says the expression is balanced. It says the expression is balanced. Now, let us try parenthesis of two different types. As you can see, what you see at the top of the stack and what you read – they are not of the same type. And therefore, you get an answer that, an expression is not balanced.

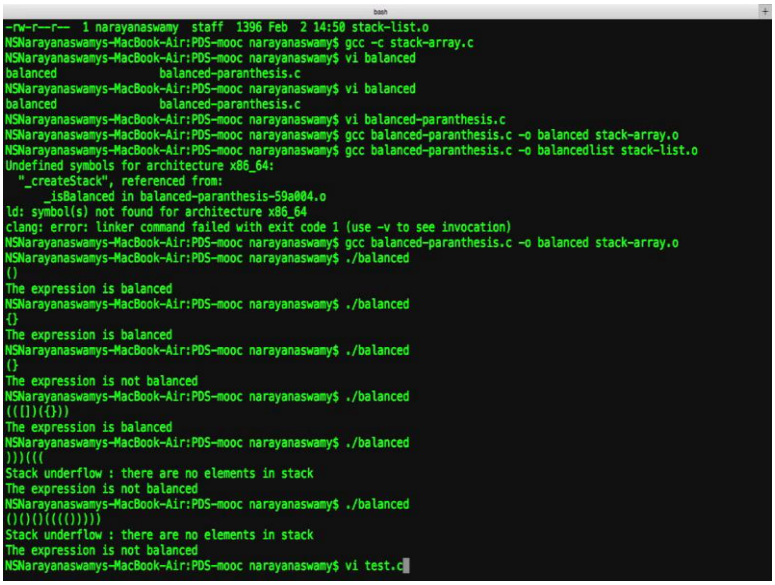
Now, let us try more sophisticated ones. I think this is balanced. And yes, it is; and that is what our program tells us. So, as you can see, we have checked quite a few of the corner cases. Let us try some other ones. How about these? There are equal number of parentheses of the same type. But, as you can see, there are no elements in the stack. So, that is because of our programming, where we do not insert these elements into the stack; we only check for the open brackets. And when the first closed parenthesis is obtained; so, we get an error saying that, the stack is empty and we immediately response – the expression is not balanced; so, in other words, just when you find a situation, where you

have one more closed bracket than the number open brackets. So, now, let us just make this a bit more interesting. If you look at this input carefully, you will notice that, I have tied one more closed bracket than open bracket. And you will see the message that, there is a stack underflow. So, the last closed bracket that I had inserted in an input was an additional closed bracket. And by then the stack had become empty. So, definitely, you make this code available to you, so that you can experiment with it.

So, really we have seen a very nice implementation of checking whether a given sequence of parentheses are balanced or not. This is a very fundamental exercise in the design of compilers; and it is repeatedly used almost by every program, every compiler that checks for the syntactic correctness of any expression or any function that you have written and so on and so forth. So, when we move to the next part of the lecture; so, you will now see a slightly sophisticated implementation – sophisticated programming exercise, where the stack gets used in a very very fundamentally different way than we have seen so far. So, what you will also observe is that, we have also used very nice programming techniques, where the interface files gave us access to different methods. We understood what the meaning of the method is and what the return type of the methods were. And using those implement… Using those methods, which are already implemented, our programming became fairly simple. So, that brings to end this first simple application of the stack data type, which is to check for balanced parenthesis.

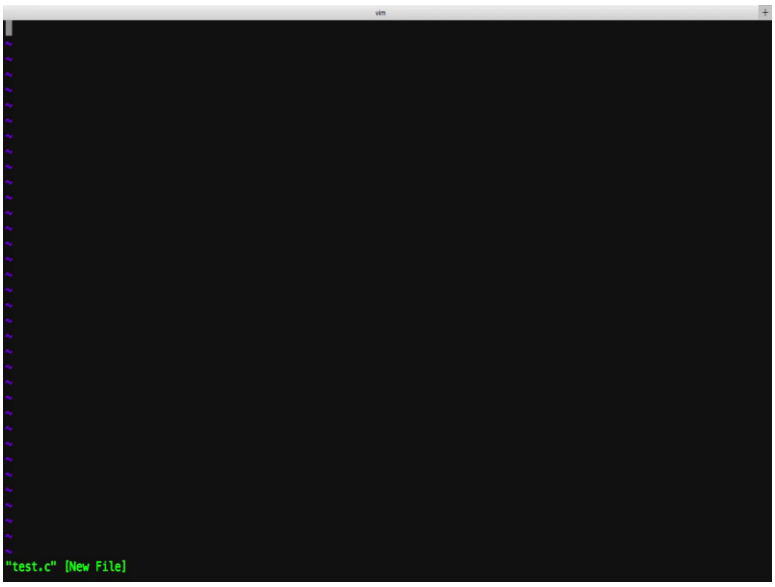
By the way, when I mentioned about compilers, it is also important for you to remember that, it is very unlikely that, a compiler just gets a sequence of parenthesis to check for whether it is balanced or not. Typically, a compiler gets an expression or a program text; or, an editor gets a program text and it has to check if the parentheses are balanced. So, you may be aware of many editors. Actually, I should show you the vi editor at this point and it is extremely useful.

(Refer Slide Time: 18:47)



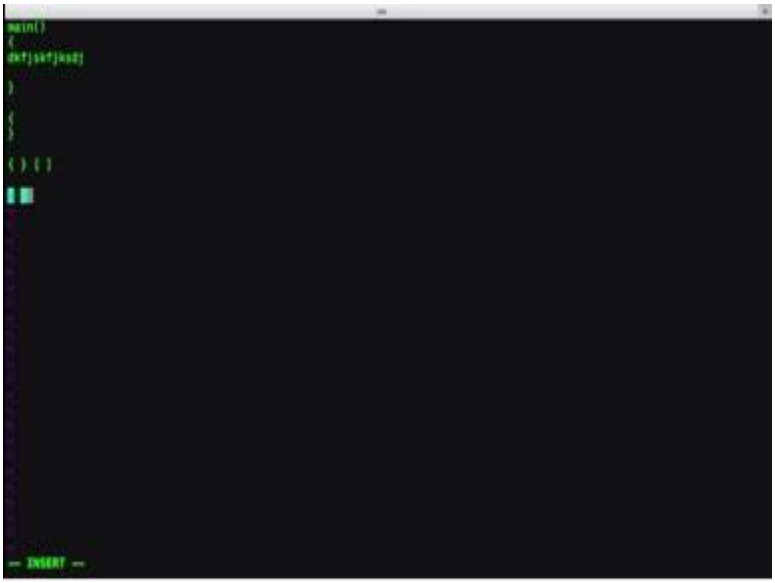
So, let me just show you this.

(Refer Slide Time: 18:50)



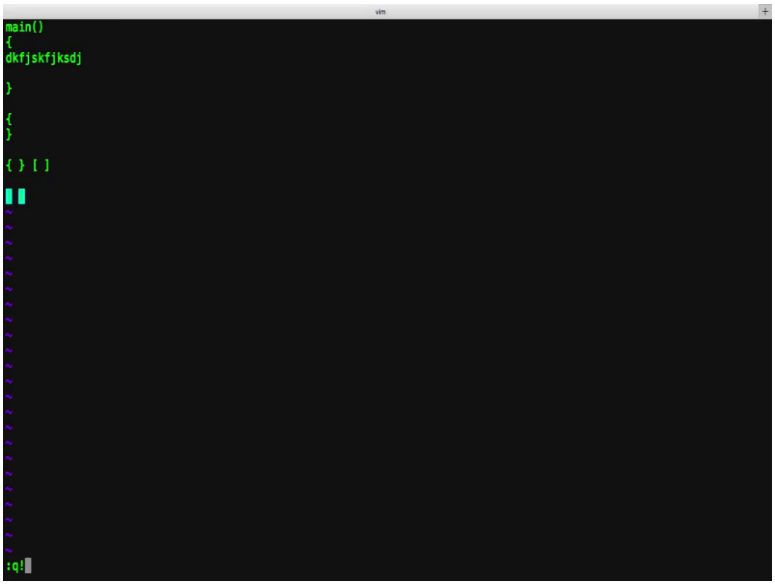
C file.

(Refer Slide Time: 18:53)



Now, you can see that ((Refer Time: 18:55)) create an open bracket. Now, I close an open bracket. Now, you see that, the editor itself is doing very sophisticated things like matching, which is the corresponding open bracket for this closed bracket. So, now, let us do this. Let us type something. And now, when I close this bracket, observe that, the editor highlights a corresponding open bracket. And this does this; does this. As you can see, it does not say anything when the parentheses are of different type. But, if it is of the same type, it immediately highlights. And essentially, what we have done is we have written a small function, which we encounter whenever we use an editor.

(Refer Slide Time: 19:47)



That should definitely be exciting to each one of you. That brings to an end this short module, which is approximately about 20 to 25 minutes on a simple, but very very important application of the stack data type. So, when we meet next, we will look at how expressions are evaluated using stacks. So, that brings to an end this lecture.