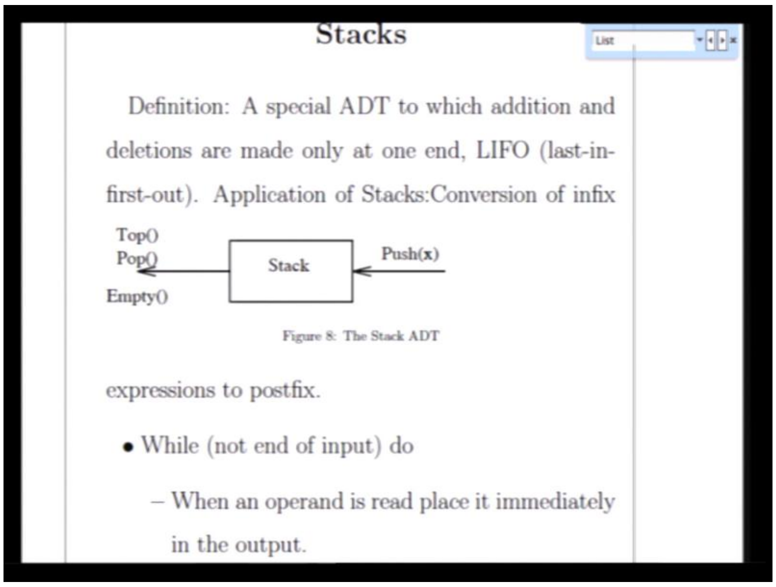
Assignment on Data Structures
Prof. Hema A Murthy
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 53

(Refer Slide Time: 00:19)



Stacks

Definition: A special ADT to which addition and deletions are made only at one end, LIFO (last-in-first-out). Application of Stacks:Conversion of infix

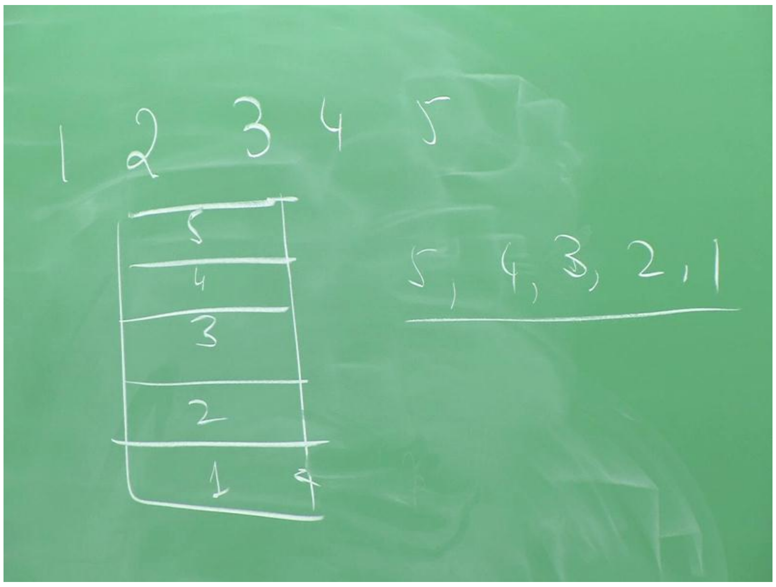
Figure 8: The Stack ADT

expressions to postfix.

- While (not end of input) do
  - When an operand is read place it immediately in the output.

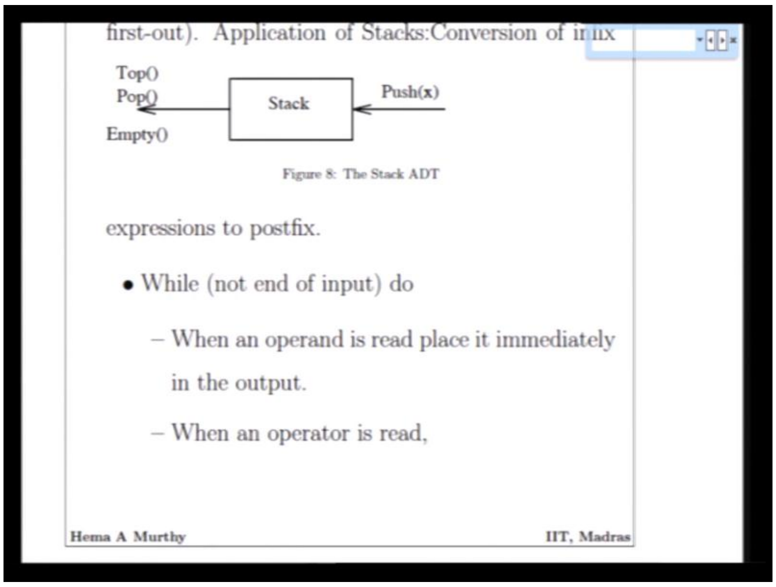
This is another supplementary video, where we try to explain to you how the conversion from infix to postfix was done using a stack, you know what is stack already we already discussed it. Therefore, you can push elements on to the stack and pop elements of the stack, you can look at the top to stack, you can check whether the stack is empty. So, basically these stack is what is called a last in last first out ADT. So, what it means is that, when we are pushing elements on to the stack and this is the stack here.

(Refer Slide Time: 00:50)



Then, let us say I push these elements we already saw this example 4 and 5, then suppose this is the top most position of the stack a pushes element here to here and so on. But, if I now pop I can only pop from here therefore, the elements if I keep continuously popping the elements will be popped in reverse order, as we saw in the example.

(Refer Slide Time: 01:23)



first-out). Application of Stacks:Conversion of infix

Figure 8: The Stack ADT

expressions to postfix.

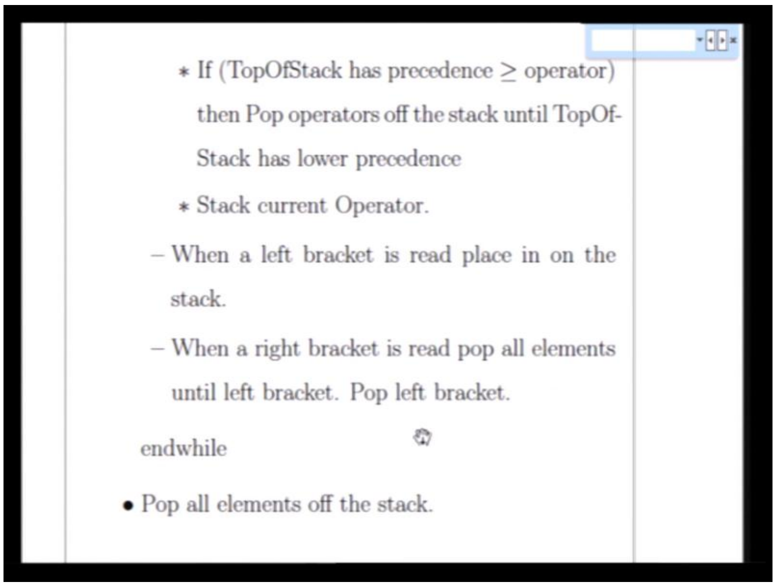
- While (not end of input) do
  - When an operand is read place it immediately in the output.
  - When an operator is read,

Hema A Murthy IIT, Madras

So, basically the operation of the stack is what is called a last in first out. And in the class know, but I did this lecture I told you how we can use this to convert a infix expression to postfix expression. What I am going to do now is, I am going to give you some C plus

plus code which does exactly that, what did we say look at the input when an operand is read place it immediately in the output.

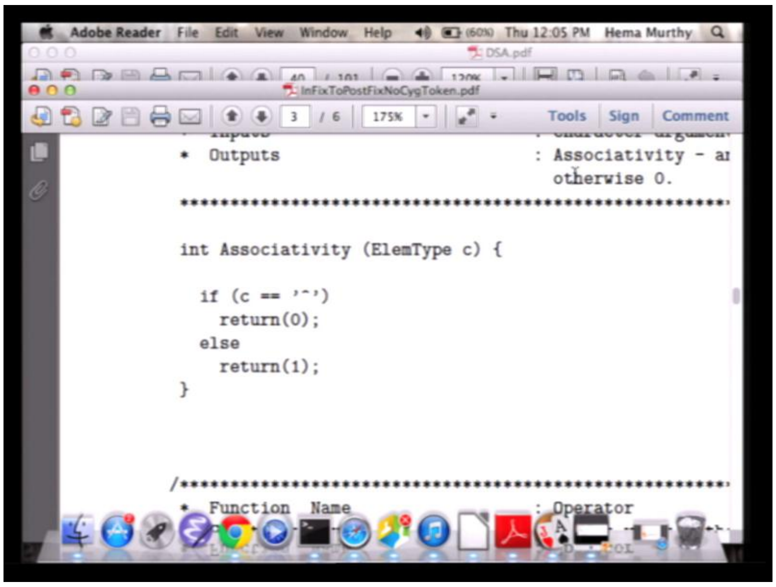
(Refer Slide Time: 01:46)



```
        * If (TopOfStack has precedence $\geq$ operator)
          then Pop operators off the stack until TopOf-
          Stack has lower precedence
        * Stack current Operator.
      – When a left bracket is read place in on the
        stack.
      – When a right bracket is read pop all elements
        until left bracket. Pop left bracket.
    endwhile
  • Pop all elements off the stack.
```

When an operator is read what did we do, if the top of the stack has higher precedence than the operator, then you pop all the operators of the stack until top of stack has lower precedence, then you stack the current operator. When you see a left bracket you place it on the stack, when a right bracket is right you pop all the elements until the left bracket pop the left bracket. Then you pop all the elements of the stack and we keep repeating this until the end of the input this pop is saw.

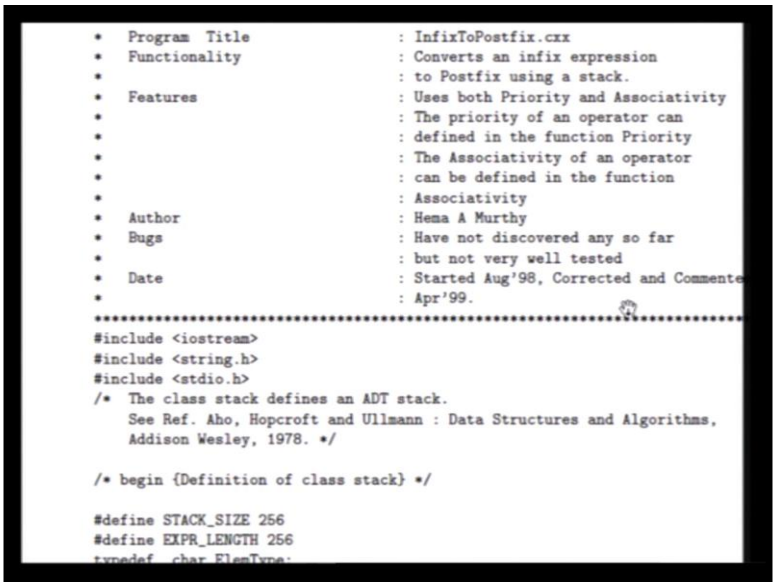
(Refer Slide Time: 02:19)



```
*  Outputs                          : Associativity - ar
                                      otherwise 0.
***********************************************************

int Associativity (ElemType c) {

  if (c == '^')
    return(0);
  else
    return(1);
}


/**********************************************************
*  Function  Name                   : Operator
```

So, now what I am going to do is, I am going to show this using a program. Let us look at this is program is in fact even more complicated.

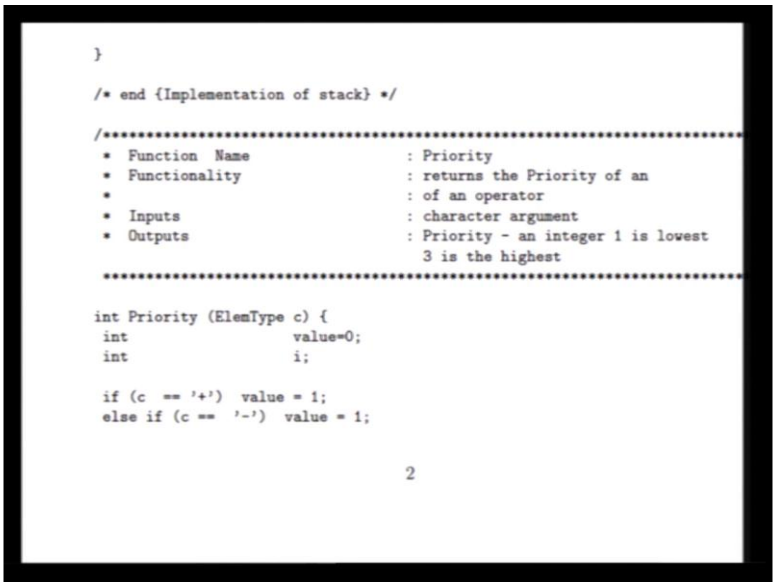
(Refer Slide Time: 02:29)



```
*   Program  Title           : InfixToPostfix.cxx
*   Functionality            : Converts an infix expression
*                            : to Postfix using a stack.
*   Features                 : Uses both Priority and Associativity
*                            : The priority of an operator can
*                            : defined in the function Priority
*                            : The Associativity of an operator
*                            : can be defined in the function
*                            : Associativity
*   Author                   : Hema A Murthy
*   Bugs                     : Have not discovered any so far
*                            : but not very well tested
*   Date                     : Started Aug'98, Corrected and Commente
*                            : Apr'99.
***********************************************************************
#include <iostream>
#include <string.h>
#include <stdio.h>
/*  The class stack defines an ADT stack.
    See Ref. Aho, Hopcroft and Ullmann : Data Structures and Algorithms,
    Addison Wesley, 1978. */

/* begin {Definition of class stack} */

#define STACK_SIZE 256
#define EXPR_LENGTH 256
typedef  char ElemType;
```

What we have done is, we have also permitted, what is called this program that we have implemented is a little more complex and let us go through this over here. So, here is the usual implementation of the stack, now I am using an array implementation of the top stack does not matter, we can use array or link list as we already saw, we already saw program which uses both and exactly runs the same way.

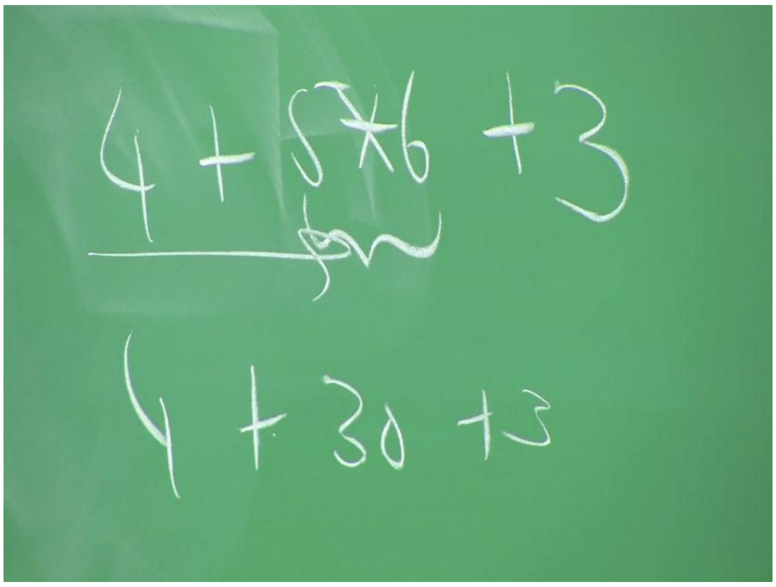
(Refer Slide Time: 03:01)



```
}

/* end {Implementation of stack} */

/************************************************************************
 *  Function  Name                  : Priority
 *  Functionality                   : returns the Priority of an
 *                                  : of an operator
 *  Inputs                          : character argument
 *  Outputs                         : Priority - an integer 1 is lowest
                                      3 is the highest
 ************************************************************************

int Priority (ElemType c) {
 int                    value=0;
 int                    i;

 if (c  == '+')  value = 1;
 else if (c ==  '-')  value = 1;

                                  2
```

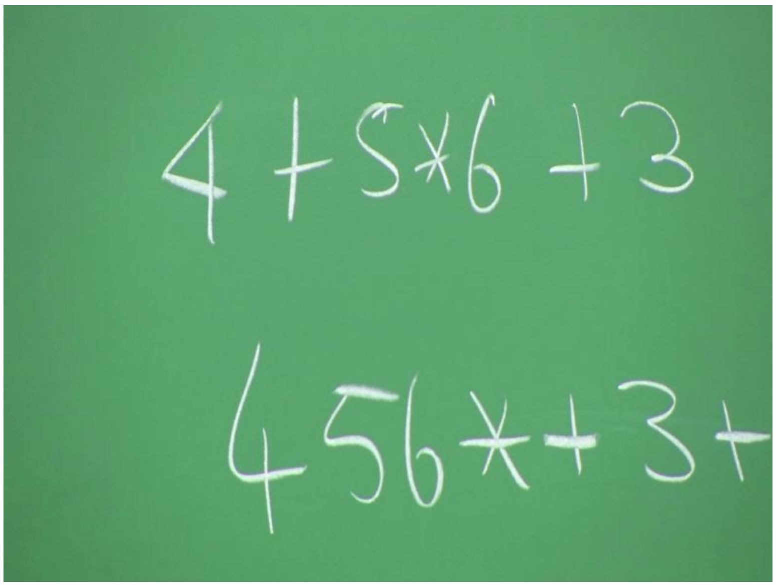
What I have done is, here is something which will give me the priority of a given, what is priority now the precedence of a particular operator.

(Refer Slide Time: 03:11)



Remember in scientific calculators when you say 4 plus 5 star plus 3 then this 5 star c will be computed first and this will be 4 plus 30 plus 3. And of course, you do the evaluation of the expression from left to right. So, we have to ensure that why do are we converting this in to postfix expression, because the postfix expression is a very convenient way of taking care of precedence's.

(Refer Slide Time: 03:41)

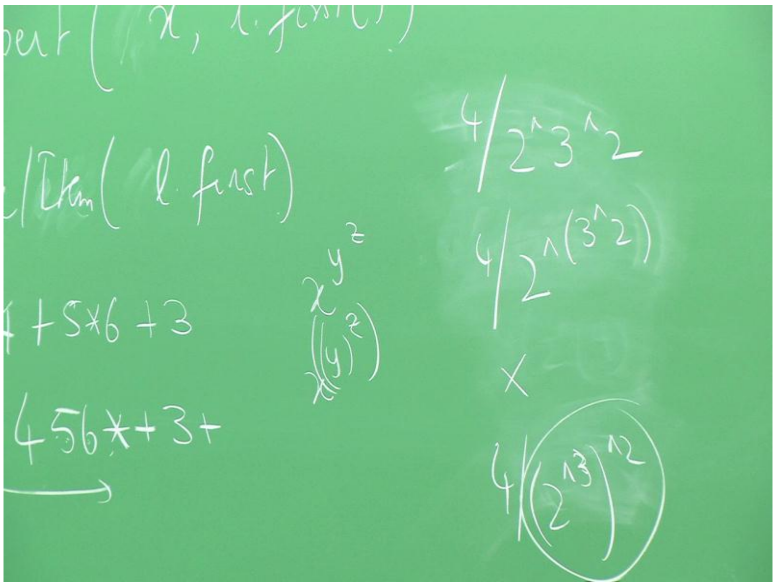


For example, if I have something like this 4 plus 5 star 6 plus 3 when I convert into postfix this will become 456 star plus and 3 plus. So, what is it mean now, if I want to evaluate this expression I keep moving from left this is the assignment that has been given to you. As soon as I come across a star, I can again use a stack for that I pop of the top most two elements on the stack and perform the evaluation.

So, when you look at this expression for example, unless you know the precedence of plus and star the relative precedence is between plus and star, you do not know which has be evaluated first. If plus is having higher precedence, this has to be evaluated first, this has to be evaluated first and then star, alternatively if here in the normal understanding of powers plus is having lower precedence. So, this has to be implemented like this.

But, as soon as we convert a postfix expression is absolutely no ambiguity, I am going to give you one more example. The another important point about postfix expressions is that even of the associativity of the operator it can be left or right.

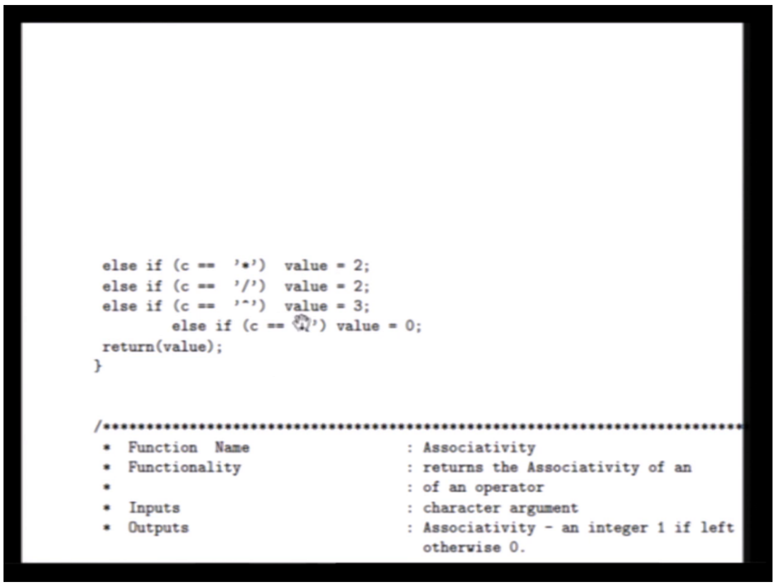
(Refer Slide Time: 05:05)



For example, we know that when you look at x power y power z, it does not it is actually x power y power z in goes like this, this how it works. So, this is called associating to the right, so when we looking at exponentiation for example, now if I have a number 4 divided by 2 power 3 power 2 this is actually 4 divided by 2 power 3 power 2 and not 4 divided by 2 power 3 power 2. This is important thing that we need to understand, such operators are what we call to the right associative, the associate to the right.

So, in this example and I am taking about there are and not do we that exponentiation also has higher priority than multiplication it is done first.

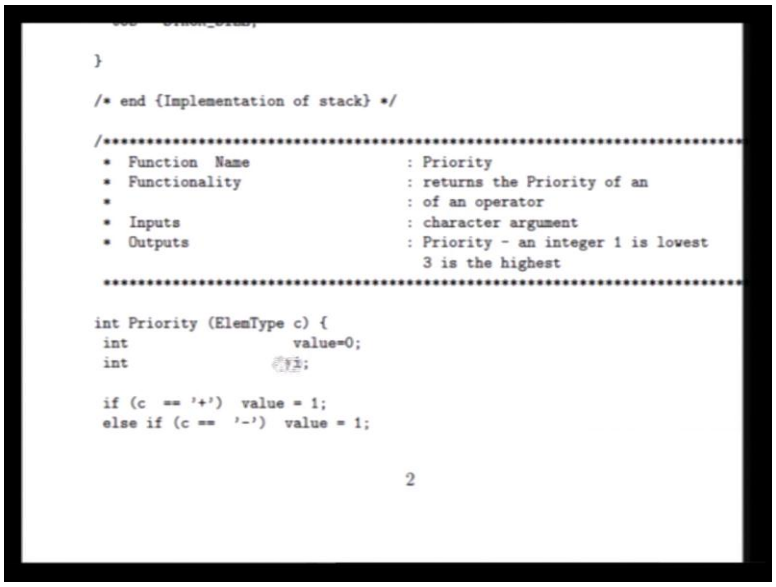
(Refer Slide Time: 06:08)



```
 else if (c ==  '*')  value = 2;
 else if (c ==  '/')  value = 2;
 else if (c ==  '^')  value = 3;
         else if (c == [illegible]') value = 0;
 return(value);
}


/***********************************************************************
 *  Function  Name                  : Associativity
 *  Functionality                   : returns the Associativity of an
 *                                  : of an operator
 *  Inputs                          : character argument
 *  Outputs                         : Associativity - an integer 1 if left
                                      otherwise 0.
```

So, there are two things that we are looking at, one is the precedence of the operators.

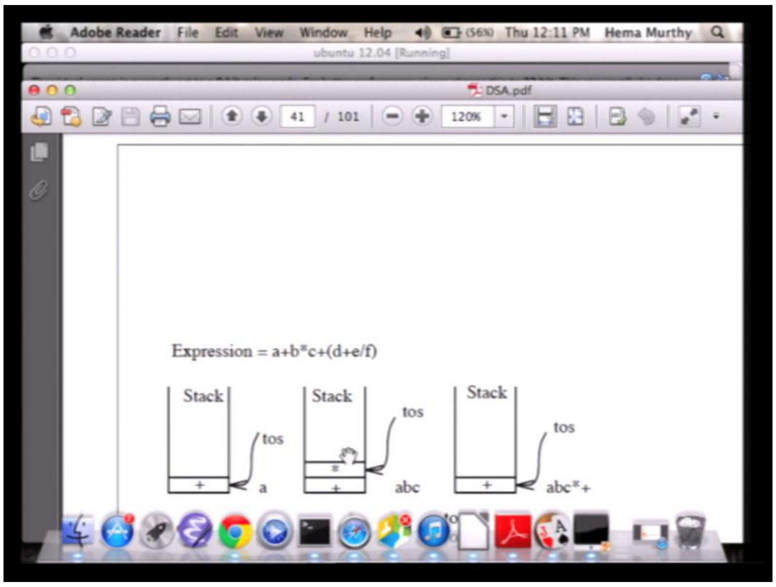
(Refer Slide Time: 06:17)



```
}

/* end {Implementation of stack} */

/***********************************************************************
 *  Function  Name                  : Priority
 *  Functionality                   : returns the Priority of an
 *                                  : of an operator
 *  Inputs                          : character argument
 *  Outputs                         : Priority - an integer 1 is lowest
                                      3 is the highest
 ***********************************************************************

int Priority (ElemType c) {
 int                     value=0;
 int                  [illegible];

 if (c  == '+')  value = 1;
 else if (c ==  '-')  value = 1;

                                2
```

So, I write one simple function which we look at the operator, which is the some particular element type and then it will tell us whether it is we know the precedence of the particular operator for example. we also put bracket, because brackets also going on to the stack as we saw in the example in the class.

(Refer Slide Time: 06:40)


Expression = a+b*c+(d+e/f)

Going back to this example in the class what it we do, if you remember right this is what we did, let us take that example now and let us say this was the expression, remember we also put the bracket on to the stack. So, this was the expression that was given a plus b star c plus d plus bracket d plus e slash f what it we do, if there is an operand with pushing it to the output an operator push it on to the stack, then we have star here then what do you do, you also again operand move it to the output star push it on to the stack operand push it to the output.

Then what happens, the next plus comes then what do you do, once a new operator comes here we did not do anything, because the star operator has higher precedence then that of plus. So, which has put it on top of it when a new operator comes I look at all the operators that are there are in the stack. So, I look at a top most way, so clearly star has higher precedence than plus therefore, we pop it off.

And then we look at next what is the on the top of the stack, the plus which is on top of stack. And clearly plus and this have the same precedence and since the operators what we call left associative, it push this plus star; otherwise we will not push it out has been for the exponentiation operation, which is given in this piece of supplementary code.

(Refer Slide Time: 07:52)

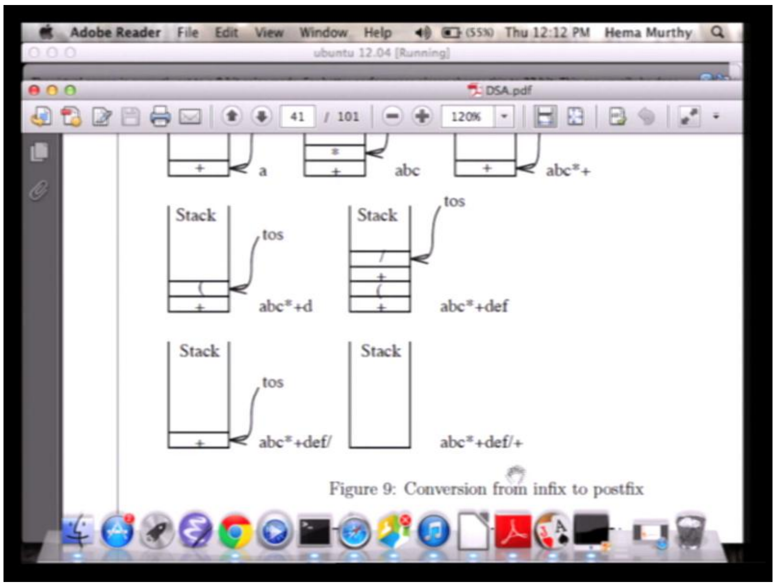
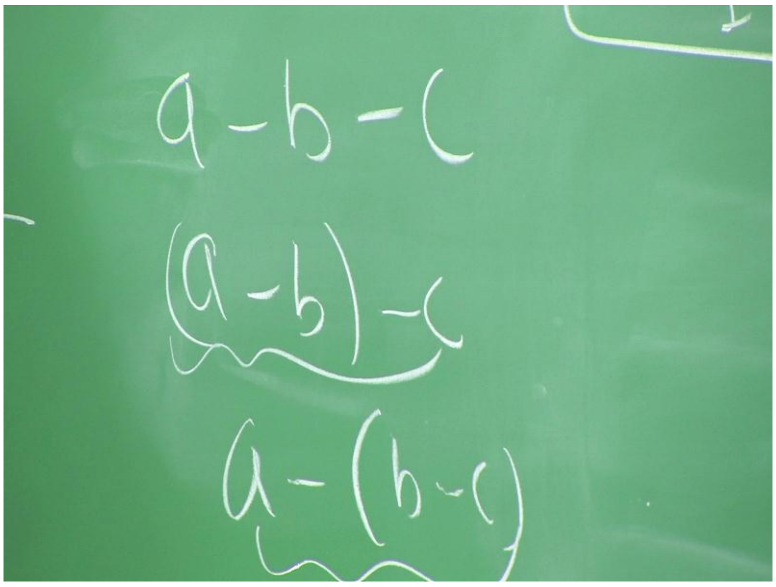


Figure 9: Conversion from infix to postfix

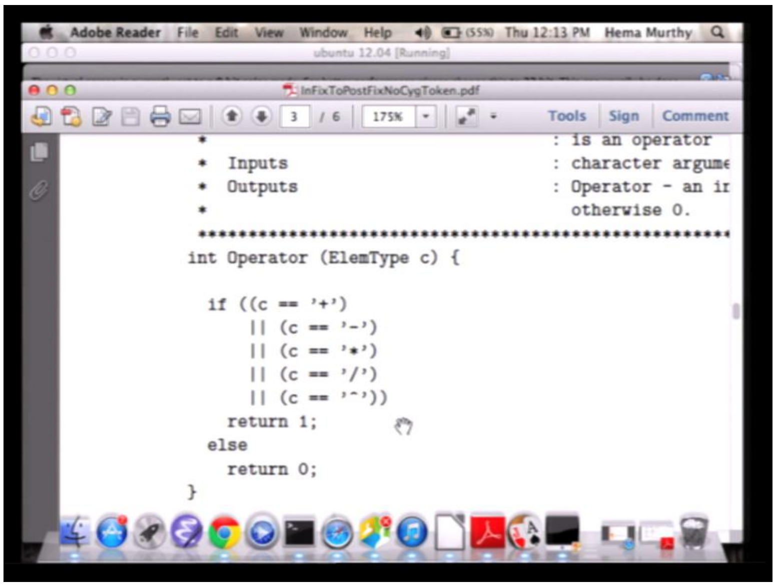
And then when a bracket comes to put the bracket and then on to the stack and then you do the rest to the operation to the usual way. And then what do you do, when is you come across the right bracket into pop everything until the right bracket, we saw this in the course. So, now, what we can to do is, so the here is a function which will return back in the priority of a given operator, I have also put exponentiation.

(Refer Slide Time: 08:30)



Then, here is another function which I have added because, so for when we look at plus and minus and multiplication and division, we always associate to the left. What I mean by this is, if I say a minus b minus c it is a minus b minus c, rather than a minus b minus c. This means association to the left and this means association to the right I said only exponentiation is different, it associates to the right as we saw some time back. So, basically what we are saying is when you doing this over here, it is 2 to the power of 3 to the power of 2, that is the fundamental difference. So, we also included that operation just to make it a little more complicated.

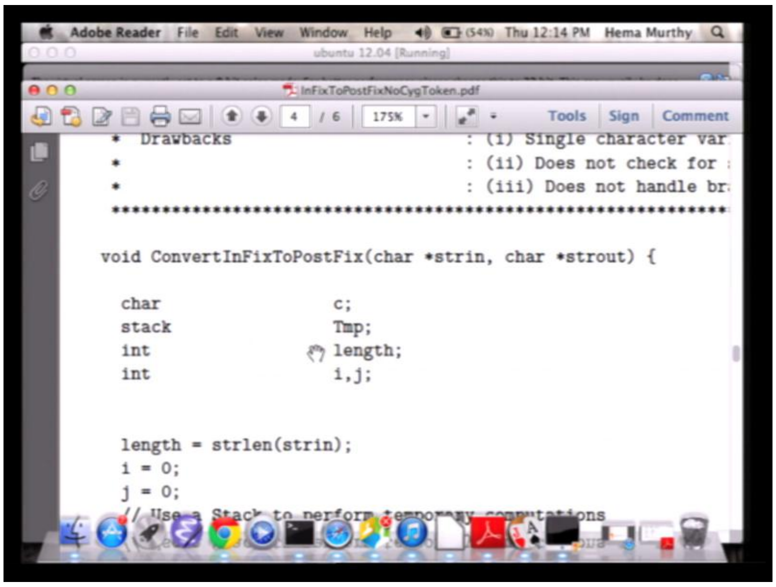
(Refer Slide Time: 09:11)



```
 *                                          : is an operator
 *  Inputs                                  : character argume
 *  Outputs                                 : Operator - an ir
 *                                            otherwise 0.
 ***********************************************************
int Operator (ElemType c) {

  if ((c == '+')
      || (c == '-')
      || (c == '*')
      || (c == '/')
      || (c == '^'))
    return 1;
  else
    return 0;
}
```

Then, what is it do here now, then we returning just for some book keeping for example, if this what is it do, this one determines the associativity. So, in associativity it is returning either 0 or 1 and then we looking at the kind of the operator, we if it is not an operator for example, then we returning a 0 over here. And now finally, let us look at a convert to infix to postfix.

(Refer Slide Time: 09:43)

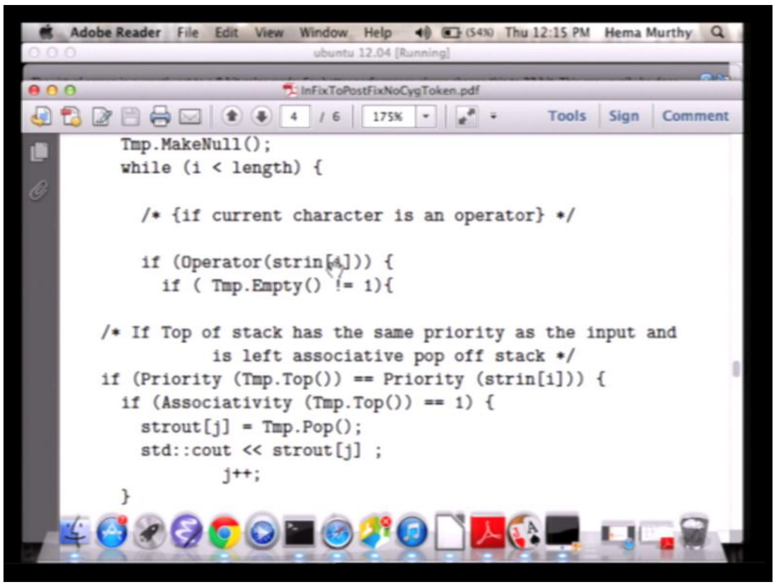


```
 *  Drawbacks                            : (i) Single character var
 *                                       : (ii) Does not check for
 *                                       : (iii) Does not handle br
 ****************************************************************

void ConvertInFixToPostFix(char *strin, char *strout) {

  char                  c;
  stack                 Tmp;
  int                   length;
  int                   i,j;


  length = strlen(strin);
  i = 0;
  j = 0;
  // Use a Stack to perform temporary computations
```

So, this is what this convert infix to postfix is a regular function like, how you would write in C or in C plus plus. What am I doing, within this convert infix to postfix we are using a stack. So, what are we doing now, so we create a empty stack, after creating a empty stack you have basically. So, what is the postfix the infix expression is given in this strin, we already saw the benefit of if this is the infix expression, this is going to be the postfix expression.

The advantage of the postfix expression is we do not have to worry about brackets, we do not have to worry about precedence. As soon as you see an operator you move from left here, as soon as you see an operator you take the most two reason guys and perform the operation, depending upon whatever you are how that is all there is to it in terms of the infix to postfix, you do not have to worry about associativity, you do not have to worry about the precedence of the operators, that is the fundamental property of it.

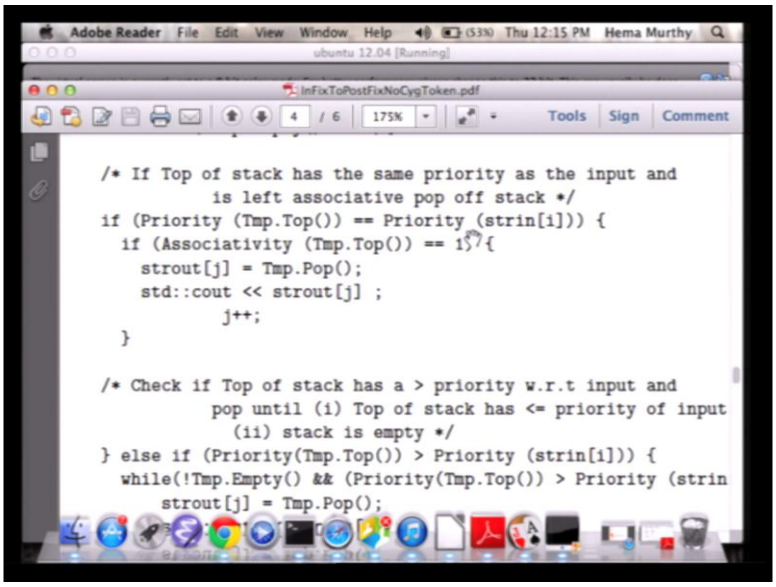
(Refer Slide Time: 10:53)



```
    Tmp.MakeNull();
    while (i < length) {

      /* {if current character is an operator} */

      if (Operator(strin[i])) {
        if ( Tmp.Empty() != 1){

  /* If Top of stack has the same priority as the input and
             is left associative pop off stack */
  if (Priority (Tmp.Top()) == Priority (strin[i])) {
    if (Associativity (Tmp.Top()) == 1) {
      strout[j] = Tmp.Pop();
      std::cout << strout[j] ;
              j++;
    }
```

So, what we will doing over here now, we check whether it is a operator, because the string is an expression that is given.

(Refer Slide Time: 11:04)

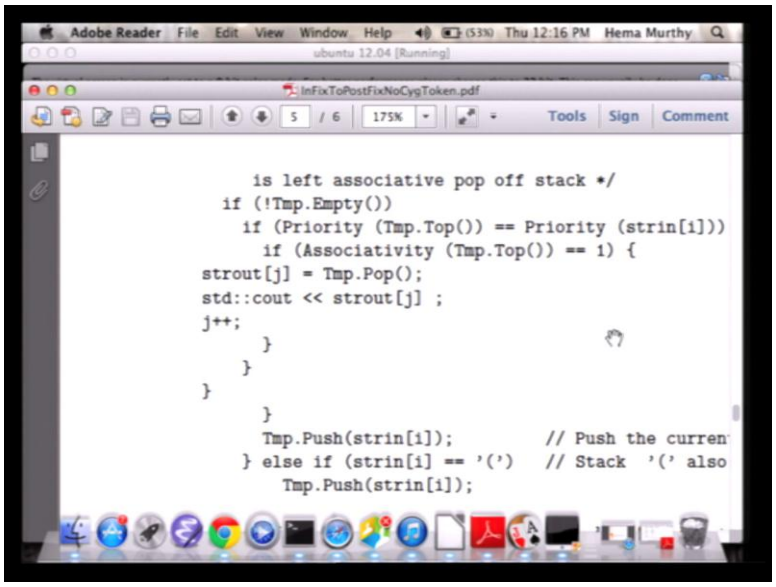


```
  /* If Top of stack has the same priority as the input and
             is left associative pop off stack */
  if (Priority (Tmp.Top()) == Priority (strin[i])) {
    if (Associativity (Tmp.Top()) == 1) {
      strout[j] = Tmp.Pop();
      std::cout << strout[j] ;
              j++;
    }

  /* Check if Top of stack has a > priority w.r.t input and
             pop until (i) Top of stack has <= priority of input
               (ii) stack is empty */
  } else if (Priority(Tmp.Top()) > Priority (strin[i])) {
    while(!Tmp.Empty() && (Priority(Tmp.Top()) > Priority (strin
        strout[j] = Tmp.Pop();
```

And then what are we doing, if the priority of the top of the stack equals to the priority of the input and if the associativity is also the same. Then, what are we doing we simply popping all the elements from the stack; otherwise, what are we doing now that is we are starting if the operator is this. Then, what we do is, if check if what are we going to do now, we need to pop elements from the top to the stack, until the stack is the priority which less than or the equal to the input or the stack is empty.

Then, we check of course, whether it is left associative or right associative that is not. So, important for you, we ensure when you do right associativity if to ensure that the element on the stack has a priority which is greater; otherwise, not equal to that is the fundamental difference.

(Refer Slide Time: 12:00)

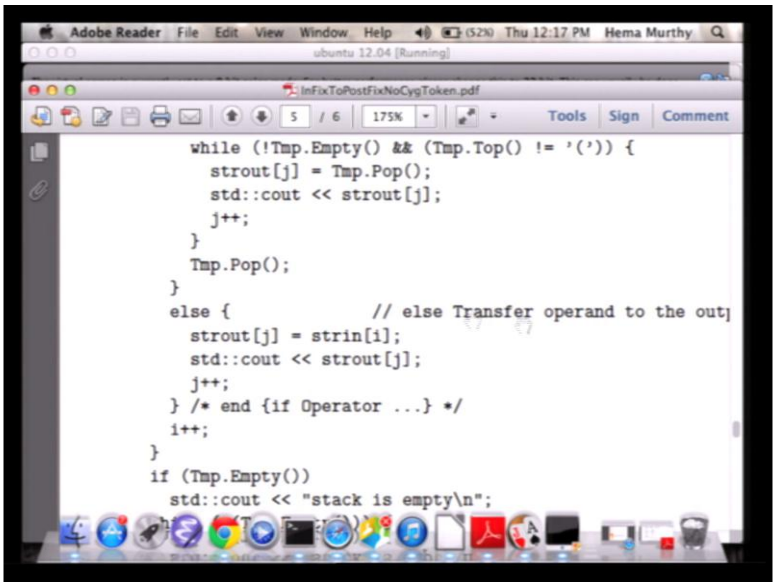


```
       is left associative pop off stack */
    if (!Tmp.Empty())
      if (Priority (Tmp.Top()) == Priority (strin[i]))
        if (Associativity (Tmp.Top()) == 1) {
strout[j] = Tmp.Pop();
std::cout << strout[j] ;
j++;
        }
      }
}
        }
        Tmp.Push(strin[i]);          // Push the curren
      } else if (strin[i] == '(')   // Stack  '(' also
          Tmp.Push(strin[i]);
```

So, in the operators are left associative we keep on popping all the elements, this is essentially the piece of code that we are looking at for infix to postfix conversion. So, basically the idea is that what I want to impress upon you all those function looks rather long is that we have an input string and then what are we doing, this input string is processed and that is we are going through this input string from left to right.

So, this is the input string is given go through left to right, if it is then what do you do, you look at this as soon as you c an element just return that, if it is not an operator then the you just simply push it to the output. And on the other hand, if it is an operand for example, if it is an operator then you check if it has the same priority as the input and is left associative or is it right associative and appropriately you decide whether you will pop the operators of the stack or not. And if on the other hand if it is an operand, you simply push the operand directly to the output, that is the fundamental activity that we are doing in this particular function.

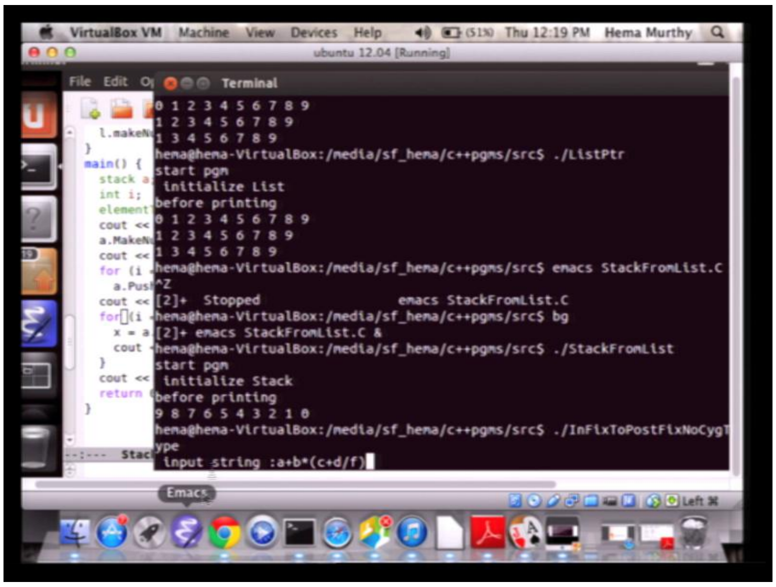
(Refer Slide Time: 13:26)



```
    while (!Tmp.Empty() && (Tmp.Top() != '(')) {
      strout[j] = Tmp.Pop();
      std::cout << strout[j];
      j++;
    }
    Tmp.Pop();
  }
  else {               // else Transfer operand to the out
    strout[j] = strin[i];
    std::cout << strout[j];
    j++;
  } /* end {if Operator ...} */
  i++;
}
if (Tmp.Empty())
  std::cout << "stack is empty\n";
```

But, what I want you to notice this here, this is where we are doing this, just it is an operand. So, I do not know have to put it onto the stack, so I simply copy it to the output string, what is interesting about this program is that. Notice that, this is again I repeat this is a normal function looks like a normal function in C, but within that it is using a stack, stack is a abstract data type. And we using the operations that are permitted on the stack, to do whatever you want and then we finally, output the particular string.

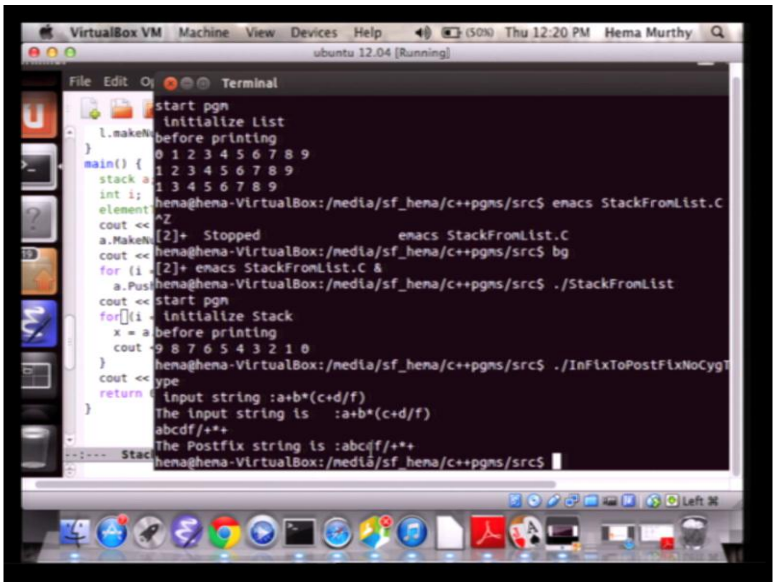
(Refer Slide Time: 14:03)



```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 3 4 5 6 7 8 9
hema@hema-VirtualBox:/media/sf_hema/c++pgms/src$ ./ListPtr
start pgm
 initialize List
before printing
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 3 4 5 6 7 8 9
hema@hema-VirtualBox:/media/sf_hema/c++pgms/src$ emacs StackFromList.C
^Z
[2]+  Stopped                 emacs StackFromList.C
hema@hema-VirtualBox:/media/sf_hema/c++pgms/src$ bg
[2]+ emacs StackFromList.C &
hema@hema-VirtualBox:/media/sf_hema/c++pgms/src$ ./StackFromList
start pgm
 initialize Stack
before printing
9 8 7 6 5 4 3 2 1 0
hema@hema-VirtualBox:/media/sf_hema/c++pgms/src$ ./InFixToPostFixNoCygT
ype
 input string :a+b*(c+d/f)
```

So, let us simply run this particular program and see what it gives you as output. So, what am I doing here now, I have this infix to postfix program and this going to run this particular function. So, it is asking me for the input string let me give a plus b star c plus d slash f whatever. So, what do we expect now, now basically there is a bracket there therefore, it has to output the elements on the bracket and the plus within the bracket should be perform first.

(Refer Slide Time: 14:54)



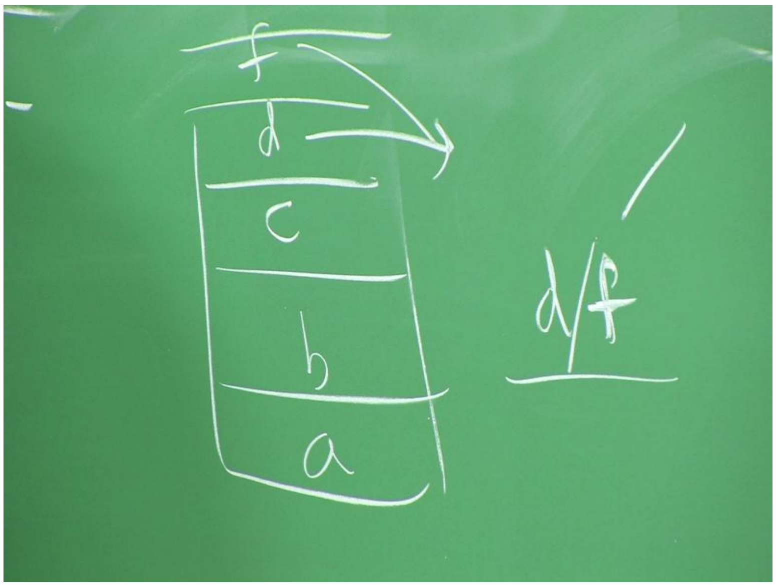
```
start pgm
 initialize List
before printing
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 3 4 5 6 7 8 9
hema@hema-VirtualBox:/media/sf_hema/c++pgms/src$ emacs StackFromList.C
^Z
[2]+  Stopped                 emacs StackFromList.C
hema@hema-VirtualBox:/media/sf_hema/c++pgms/src$ bg
[2]+ emacs StackFromList.C &
hema@hema-VirtualBox:/media/sf_hema/c++pgms/src$ ./StackFromList
start pgm
 initialize Stack
before printing
9 8 7 6 5 4 3 2 1 0
hema@hema-VirtualBox:/media/sf_hema/c++pgms/src$ ./InFixToPostFixNoCygT
ype
 input string :a+b*(c+d/f)
The input string is   :a+b*(c+d/f)
abcdf/+*+
The Postfix string is :abcdf/+*+
hema@hema-VirtualBox:/media/sf_hema/c++pgms/src$
```

So, we do this and this gives us a b c d f notice; that means, the first operation is going to be perform is d slash f and then it performs this plus and then the star and finally, the plus. So, now, we one of the assignment problems is to perform postfix evaluation using the stack. So, what is going to happen now, earlier you had be the element type on the stack which consisted of characters which corresponded to your operators.

Now, the element type on the stack will be integer type. So, what are we doing now, you given a string, so what you have to do is given a string you have to perform the assume that you are given a postfix string and you are suppose to evaluate the postfix string I thing we that is basically the idea. So, you can again use a stack here is as obvious because d and f are operands here.

So; that means, I just keep on pushing all the elements on to the stack. So, what happens still I come across an operator when I come across an operator what do I do. So, on my elements of the stack I have a b c d e f I take of the top most two elements perform the operation and put it back on the stack. When I see a plus then what do I do, I take of the top most two operands pushed on to the stack, let us look at with an example over here.

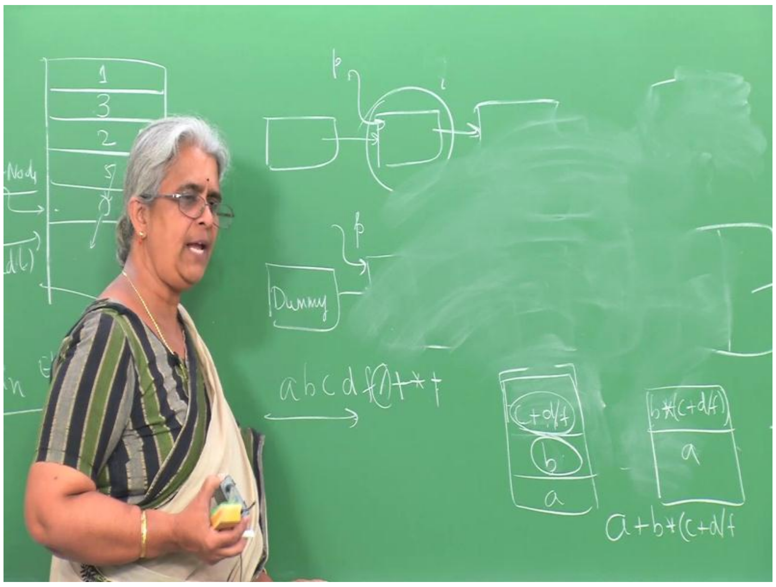
(Refer Slide Time: 16:36)



So, what it that we are say if this is the stack at push a, b remember this is the post fixed expression. That means, we are giving a, b, c, d, f slash plus star and plus that is what we have as the postfix expression. So, what are we doing now on to the stack now, if this is our stack we simply push all the elements. That means, I am reading this post fixed expression from left to right, I push it on to the stack d, f now what happens I am seeing an operator.

So, as soon as I see an operator I pop these two elements from this stack, it perform because it is from left to right and performing d slash f. I perform d slash f then what do I do, I push it back on to the stack d slash f. Now, next what do I see, I see another operator plus when I see the other operator plus what do I do, I pop these two elements and what do I do I have C plus, plus is the operator. So, p plus d slash is the f is perform.

(Refer Slide Time: 18:04)



Again you push it on to the stack, next you see a star when you see a star what it will be doing now. Again we perform b star, because these are the c plus b slash f is there and b star therefore, this is multiplied with this contained, again you push it back on to the stack. So; that means, what are we doing now, we have b plus c plus b star c plus d slash f push it on to the stack and then you have a, you have a minus or plus it is a plus over there.

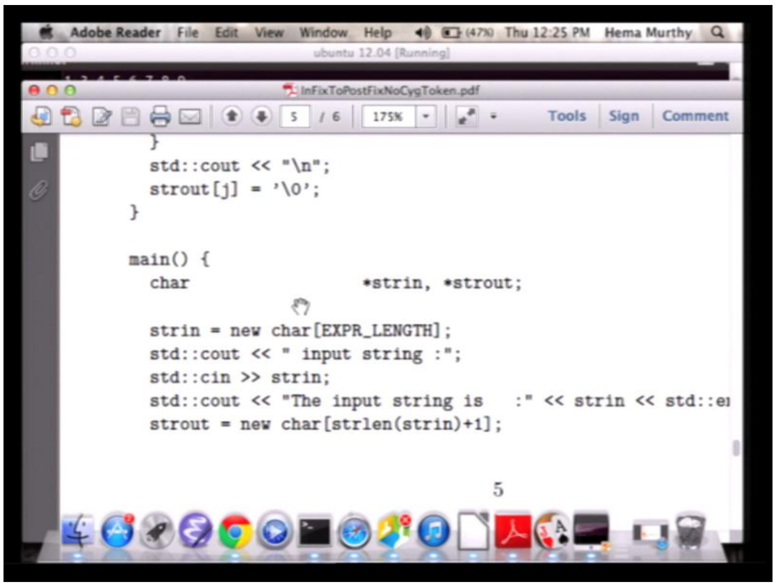
So, you have plus here and then you do a plus b star c plus d slash f and the stack becomes empty, there are no more. Notice that, the input string is over and you just simply output the stack is also empty, because what happens I remove these two elements and then I am simply putting the result. Therefore, the result consist of the is corresponds to the evaluation of the expression. So, this is exactly what you are expected queue on the assignment.

What I have shown you here is infix to postfix conversion using the stack over here. So, what it we do, we just giving the input string here and then given that this is the input string, it converts is to the postfix string over here. So, this is primarily your task on the second part of the assignment. So, I hope by now you have understood two important things, what are the two important things that we saw, in this lecture we saw that you can use a stack to perform infix to postfix conversion.

And if you remember right notice that over here, this function infix to postfix which is there in the code over here, notice this over here. So, this convert infix to postfix is something that we are not get c in before, convert infix to postfix is another function. And what is I do, it uses a stack internally a stack has already been defined and the element type it goes on to the stack is what we user specifies.

And then he simply using this stack like, you know that plus will if I give two floating point numbers plus will add to floating point numbers, here is stack will push elements on to the stack pop elements from the stack is a last in first out ADT. And that precisely the property of the ADT, that the user uses here is the convert infix to postfix function which is doing this particular operation.

(Refer Slide Time: 20:44)



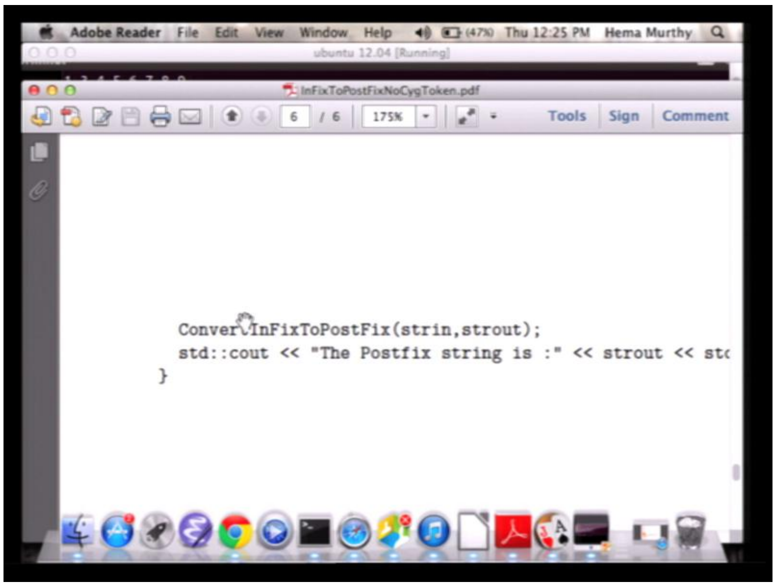
Adobe Reader File Edit View Window Help (47%) Thu 12:25 PM Hema Murthy
ubuntu 12.04 [Running]
InFixToPostFixNoCygToken.pdf
5 / 6
175%
Tools
Sign
Comment

```
    }
    std::cout << "\n";
    strout[j] = '\0';
  }

  main() {
    char                    *strin, *strout;

    strin = new char[EXPR_LENGTH];
    std::cout << " input string :";
    std::cin >> strin;
    std::cout << "The input string is   :" << strin << std::e
    strout = new char[strlen(strin)+1];
```

5

And then notice I have another name program and top of it, where what do you do you have define some particular expression link, that is the users idea. So, I gives the particular expression line, inputs the string the string is read and then what is we do, he again creates another. So, given the length of the string it creates a new output string and then passes this two, passes string in and string out.

(Refer Slide Time: 21:15)



```
  ConvertInFixToPostFix(strin,strout);
  std::cout << "The Postfix string is :" << strout << stc
}
```

Notice that, this whole main program is completely harmless main program does not know that convert infix to postfix is actually using a stack inside. So, this is something that is very important to understand, so we have seeing primarily two different characteristics over here, we have seen the stack ADT implemented with the particular element type.

And we have seen the convert infix to postfix for example, is actually using the stack ADT inside, the main program is not aware of it. So, this is the important take a way and in your assignment for example, in post fixing evaluation what are you expect to do, is not a stack of operators, it is a stack of operands that was given to you. So, given that what we have done is, in that particular assignment for example, the stack has been implemented, the implementation may not be exactly the way I have done it in the course it can be a little bit different.

But, it still has the pop the top and empty operations, that is kind of irrelevant to you how it is be implemented, using that stack how can you implement postfix evaluation that is the question that is asked. And we see now what is it that we have to do, as you see an operand every operand is pushed onto the stack and an assure as you see an operator, you perform the evaluation. Notice the fundamental difference between convert infix to postfix and postfix evaluation. In convert infix to postfix what are we doing, we pushing the operators on to the stack, in postfix evaluation we are pushing the operands on to the stack.