

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

Lecture – 11
Infix and Postfix Expressions and Expression Evaluation

So, let us look at the third lecture on the stack abstract data type and let us look at one more application of the stack abstract data type. Recall that in the last lecture, we looked at using stacks for checking balanced parenthesis. And we also saw at the end of the lecture that it was not a toy programming exercise; it is a very, very important programming exercise. I showed you the editor that I use on my computer that highlights matching parenthesis. And we essentially wrote almost that whole program. In today's lecture, what we are going to do is we are going to look at expression evaluation. So, we are going to see that expression evaluation, which involves precedence of arithmetic operators is fairly sophisticated if one has the expression in the format that we are comfortable with. This is the infix format; where for every operator, the operands occur to the left and right respectively. For every binary operator, the operands occur to the left and right respectively. So, we have very sophisticated precedence rules there especially when there are parentheses. So, most of you will recall that there are these rules called BODMAS rules, which tell you in what order you must evaluate sub-expressions to be able to correctly evaluate a given expression.

Now, we will also see that, if the expression is given a slightly modified form; then the expression evaluation rule becomes very simple. And therefore, the challenge is going to be to convert the expression from a given form, which is the infix form; where the operators are preceded and succeeded by their operands. Conversion from this format to the simpler format, which I call the postfix is going to be a very interesting programming exercise. So, that is the focus of today's lecture. So, we will spend about again 20 minutes on this exercise – 20 to 25 minutes on this exercise, where we will look at the slides to describe the programming exercise and then quickly go to the program that I have written.

(Refer Slide Time: 02:28)

Mathematical Calculations

- What is $3 + 2 * 4$? $2 * 4 + 3$? $3 * 2 + 4$? - The precedence of operators affects the order of operations. A mathematical expression cannot simply be evaluated left to right.
- What about $1 - 2 - 4 ^ 5 * 3 * 6 / 7 ^ 2 ^ 2$
- How does one represent the expression so that precedence rules can be easily checked?
- The problem is that for each operator, the two operands which occur to the left and right of the operator are not directly available from the expression - **INFIX**
- So, can we rewrite the expression to get an equivalent expression in which the operator follows the two operands? - **POSTFIX**

So, we are interested in looking at mathematical calculations like I said. For example, we can ask this question – what is 3 plus 2 multiplied by 4; what is 2 multiplied by 4 plus 3; and what is 3 multiplied by 2 plus 4. And if we do not have a uniform convention about which operators will be evaluated first; then what can get different answers. For example, let us look at this simple one. Most school children are likely to think of this as 5 multiplied by 4; that is, 3 plus 2 multiplied by 4; if you ask a third standard kid, who understands bit of arithmetic; that child is very likely to tell you that, it is 5 multiplied by 4, which is 20. But, the minute we put in the conventions of the precedence rules, then the answer we all know is actually 11; that is, the multiplication needs to be done first and followed by the addition; so that is, this multiplication needs to be done first followed by the addition.

This can get even more complicated in the presence of parenthesis; but before that, we can have also other binary operators like exponentiation. Recall that, exponentiation is a binary operator; 4 to the power of 5. Similarly, here you will see 7 to the power of 2 to the power of 2. Now... So, this is 7 to the power of 4 basically. And these are all... So, we need evaluation rules. So, for example, exponentiation has the highest priority followed by multiplication and division; but multiplication is done first; followed by this multiplication; followed by this division. So, therefore, in the denominator, will be 7 to

the power of 4. And this exponentiation is also done first and so on and so forth. So, therefore, how does one represent the expression, so that the precedence rules can be easily checked.

Now, what is the problem? The problem is that, the two operands that occur to the left and right of an operator are not directly available from the expression. In other words, because of our precedence rules, if you ask what are the arguments, what are the operators for this minus sign? We know from our evaluations that, it is the result of 1 minus 2 followed by the result of this whole expression. So, it is really 4 power 5 multiplied by 3 multiplied by 6 divided by 7 power 4. The result of that is subtracted from the result of 1 minus 2. Therefore, that is what is brought out in this point. We say that, for every operator, the two operands, which occur to the left and right, are not immediately available. For example, when I mean immediately available; for this minus, you cannot say the operands are 2 and 4 respectively; you can only say that, it is the result of the expression 1 minus 2. And the second operand is the result of this very complicated looking expression. Therefore, how does one extract this information is the challenging question.

So, now, the question is – can rewrite the expression, so that we can get an equivalent expression in which the operator follows the two operands. So, therefore, let us just see this. Is it possible to rewrite the expression, so that in a simpler form; that is, whenever you see an operator, the immediate two symbols preceding it are essentially are the operands. So, let me restate the sentence again. Is it possible to rewrite a given expression; that is, a given expression in this form. In an equivalent format, where every operator follows its two operands; such a notation is called the postfix notation. The notation that we are used to from schooldays is called the infix notation.

(Refer Slide Time: 06:50)

Infix and Postfix Expressions

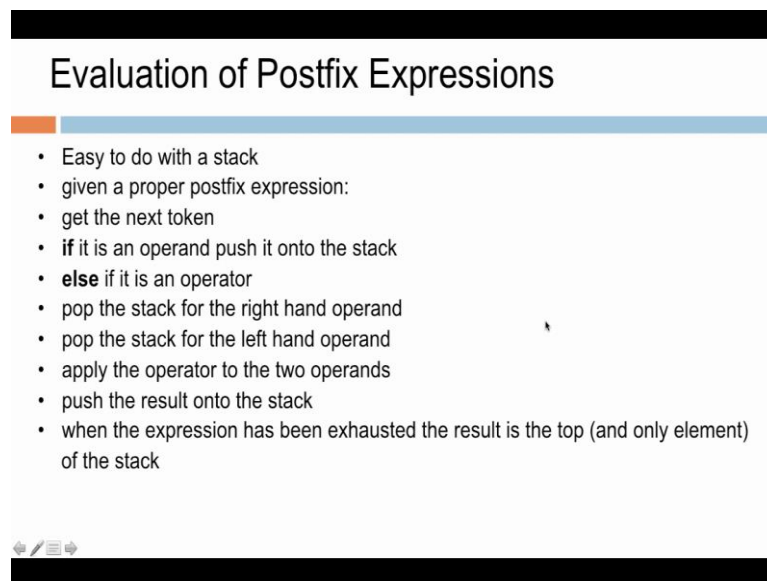
- $3\ 2\ *\ 1\ +$ is postfix of $3\ *\ 2\ +\ 1$
- Postfix Expression
 - No precedence
 - Read the expression from left to right.
 - When you see an operator, take the two operands and evaluate
- $2\ 3\ 2\ 4\ *\ +$ is postfix of $2\ * (3 + 2*4)$
- $2\ 3\ *\ 2\ 4\ *\ +$ is postfix of $2\ * 3 + 2\ * 4$
- $1\ 2\ 3\ 4\ ^\ *\ +$ is postfix of $1 + 2\ * 3^4$
- Seems simple to evaluate postfix, is it not?
- Let us look at an algorithm to evaluate postfix expressions
- Following that, How does one convert from Infix to Postfix.?

So, let us look at a few examples of the infix and postfix expressions. Let us look at the postfix expression $3\ 2\ -$ – these are the two operands followed by star followed by 1 followed by plus. So, now, this is the postfix of $3\ star\ 2\ plus\ 1$. So, the postfix expression really has no precedence. All that you have to do is read the expression from left to right. Whenever you see an operator, take the two operands and evaluate it. ((Refer Time: 07:27)) we will see how to do this in a short while; but the rule is very simple. Whenever you see an operator, you pick the two operands that occur just before it. For example, you may be confused to find out what are the two operands for this plus; one operand is 1; the second operand is obtained by evaluating the star. What are the operands for this star? It is 2 and 3 respectively. Therefore, the operand for this plus is 1 and 6. Therefore, the result will be a 7 and so on and so forth. Therefore, let us go through the rules for evaluating a postfix expression; we just saw this. Let us do another one here.

Let us look at $1\ 2\ 3\ 4\ ^\ *\ +$ exponentiation star and plus. So, let us scan the expression from the left to right. So, you see – $1\ 2\ 3\ 4\ ^\ *$ exponentiation star plus. So, now, when you see the exponentiation, what are the two operands for the exponentiation? 3 is a left operand and 4 is the right operand. So, you see 3 to the power of 4. So, therefore, you will see 3 to the power of 4 now replacing this fellow. Now, you see a star. What are the operands for star? The left operand... The right operand for star is 3 to the power of 4; and the left

operand is the 2 as you see here. Finally, you see a plus. The two operands for plus are the right operand is the result of 2 multiplied by 3 to the power of 4; and the left operand is a 1. Therefore, evaluating postfix seems to be fairly simple. So, let us first look at the few examples for evaluating postfix expression; and following that, let us see how to convert from infix to postfix.

(Refer Slide Time: 09:23)



Evaluation of Postfix Expressions

- Easy to do with a stack
- given a proper postfix expression:
- get the next token
- **if** it is an operand push it onto the stack
- **else** if it is an operator
- pop the stack for the right hand operand
- pop the stack for the left hand operand
- apply the operator to the two operands
- push the result onto the stack
- when the expression has been exhausted the result is the top (and only element) of the stack

So, evaluation of postfix expression is very simple. We will do this using a stack. Given a proper postfix expression, we get the next token, which is the next symbol. I call this a token because when you look at a program and if you are identifying the order in which expression must be evaluated, a variable may not just be a single symbol; but it may be a combination of multiple symbols. Now, if it is an operand; that is, if it is an arithmetic... If it is an operand, you push it on to the stack. If it is an operator, you pop the stack first to get the right-hand operand; then you pop the stack to get the left-hand side operand. And you apply the operator on the two operands and push the result back on to the stack. And when the expression has been done, whole expression is exhausted; the result will be at the top of the stack. So, let us go through this again. You pass or you go through the whole expression from left to right. Whenever you find an operand, you push it on to the stack. If you find an operator, pick that two operands from the top of the stack; perform the arithmetic operation; push it back on to the stack; repeat till the whole expression has

been processed. That will essentially give you a way of evaluating the postfix expression.

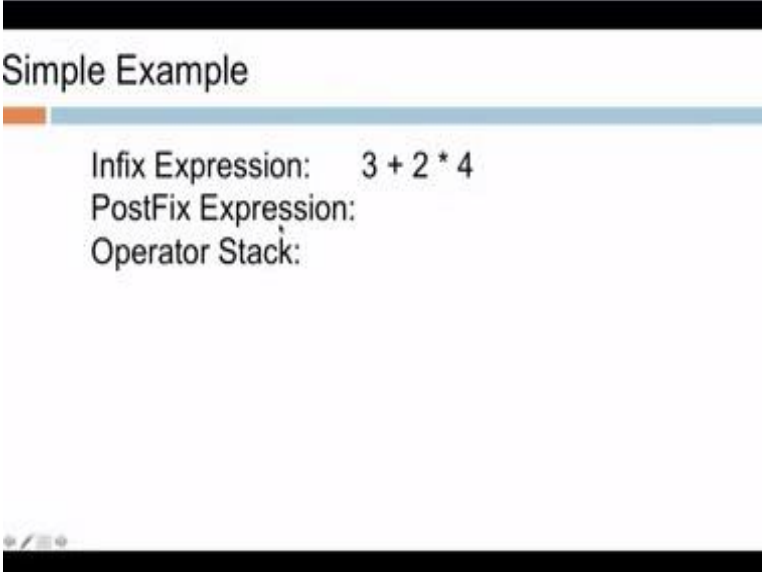
(Refer Slide Time: 10:53)

Infix to Postfix

- $2^3 \cdot 3 + 5 \cdot 1$ is the infix of $2\ 3\ 3^{\wedge}\ 5\ 1\ *\ +$
- $11 + 2 - 1 \cdot 3 / 3 + 2^2 / 3$ is the infix of $11\ 2\ +\ 1\ 3\ *\ /\ -\ 2\ 2\ ^\ /\ +$
- Key issue, the precedence of operators must be enforced.
- Let us see a simple example next.
- How does one deal with parenthesis in the expression?

We will definitely look at a program, which does this. And let us look at now the challenging exercise of converting an infix expression into a postfix expression. For example, let us look at the infix expression – 2 power 3 power 3 plus 5 star 1. So, clearly, what are supposed to do? We are supposed to take 2 power 3, which is 8; and 8 to the power of 3; and then add it to the result of 5 multiplied by 1. This is the application of the precedence rules. So, this is the infix expression for this postfix expression. So, now, let us just see whether we get exactly the same procedure or this expression when we apply the algorithm that we saw on the previous line. So, we push 2 on to the stack, 3 on to the stack, then 3 on to the stack; then exponentiation comes. So, when exponentiation comes, we take the previous two operators, which is now gives us 3 power 3; and which is to write it on to the stack. And then we considered 2 power 3 power 3 and then we get to compute the product of 1 and 5 and then we get to add the whole result. Similarly, you can see that, this is the infix of this particular expression. The key issue here is that, somehow, in the conversion of the infix to the postfix form; the precedence of the operators must be some how enforced. And we also will have to come up with a method to deal with paranthesis in the expression. Before that, let us look at...

(Refer Slide Time: 13:07)



Simple Example

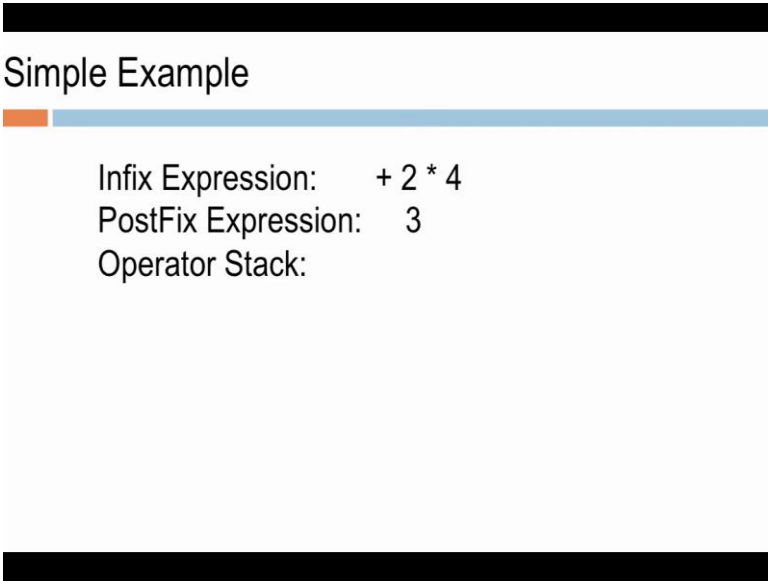
Infix Expression: $3 + 2 * 4$

PostFix Expression:

Operator Stack:

Before we go to the program, let us look at a simple example. Let us look at the example and then we will come back to the... So, even as the infix expression, 3 plus 2 star 4; let us look at the postfix expression in the operator.

(Refer Slide Time: 13:17)



Simple Example

Infix Expression: $+ 2 * 4$

PostFix Expression: 3

Operator Stack:

So, the operand comes. Now, we write it into the output postfix expression.

(Refer Slide Time: 13:27)

Simple Example

Infix Expression: $2 * 4$
PostFix Expression: 3
Operator Stack: +

Now, the operator comes; it is pushed on to the stack.

(Refer Slide Time: 13:30)

Simple Example

Infix Expression: $* 4$
PostFix Expression: 3 2
Operator Stack: +

Then, another operand comes; we write it into the expression.

(Refer Slide Time: 13:46)

Simple Example

Infix Expression:
PostFix Expression: 3 2 4
Operator Stack: + *

Then, we push the next operator on to the stack.

(Refer Slide Time: 13:51)

Simple Example

Infix Expression:
PostFix Expression: 3 2 4 *
Operator Stack: +

Then, we print out the last term.

(Refer Slide Time: 13:53)

Simple Example

Infix Expression:
PostFix Expression: 3 2 4 * +
Operator Stack:

And then we pop from the stack while it is empty.

(Refer Slide Time: 14:01)

Infix to Postfix Conversion

- Maintain **operators** in a stack- to push or pop depends on the precedence with respect to the operator at the top of the stack!!
- Operands: add to output postfix expression
- When you see an open parenthesis, push onto stack.
- Close parenthesis: pop stack symbols until an open parenthesis appears
- When you see an Operator:
- **Pop** all stack symbols **until** a symbol of lower precedence appears. **Then** push the operator on to the stack
- When you pop, you print them into the output postfix expression
- End of input: Pop all remaining stack symbols and add to the expression

So, that was just a very simple example. And let us look at the complete algorithm for achieving this. We will exactly implement this piece of code that you see here in the C program, which we will see shortly. So, here what we do is we maintain the operators in

a stack whether to push the operator on to the stack or pop from the stack depends upon the precedence with respect to the operator at the top of the stack. So, let us just understand this. When I see certain operator, to decide whether to push on to the stack or pop on to the stack depends upon the precedence with respect to the operator at the top of the stack. So, we will see this rule in a short while. The operands – whenever we see an operand, we send the operand to the output postfix expression directly.

Now, let us look at the parenthesis and the operators. The first rule is when you see an open parenthesis; you push it on to the stack. When you see a close parenthesis; you pop all the stack symbols and print it out one after the other. Remember the stack symbols are all the operators; you pop them all out until an open parenthesis appears; and you pop the open parenthesis and print it out. When you see an operator, the rule is as follows. You pop all the stack symbols until the top of the stack consists of an operator of a lower precedence. Then you push the operator on to the stack. Let us go through this again. When you see an operator; you pop all the stack symbols until an operator of lower precedence is obtained at the top of the stack. When you see this operator of lower precedence, you then push this operator on to the top of the stack.

Understand what we are seeing. When you see a certain operator, what we are basically saying is that, we will perform on the stack, all the operations of higher precedence first and then we will perform this operator and then we will perform an operator of the lower precedence. That is the idea here. So, you pop all stack symbols until you see a symbol of lower precedence. Then you push the operator on to the stack. Anyway whenever you pop anything, you print it into the output postfix expression. At the end of the input, you pop all the remaining stack symbols and add it to the expression. So, this is the whole idea of the infix to postfix conversion algorithm. So, observe this. The input expression, which is an infix form has three kinds of symbols: operators, operands and parenthesis. Whenever you see a parenthesis – open parenthesis, you push it on to the stack. When you see a close parenthesis, you pop all the operators on the stack till you see an open parenthesis, which also you will pop out.

And of course, you never print out; I have not return this here; that was a small mistake on my side. Remember that we will never print out the parenthesis in the postfix

(Refer Slide Time: 18:27)

Let us go to the program, which implements the infix to postfix convertor.

(Refer Slide Time: 18:29)

```

AllistMethods.c
AllistMethods.o
lecture 1.pptx
list-interface.h
list.h
listMethods.c
listMethods.o
MOOC_Syllabus_template.doc
MOOC_Syllabus_template.odt
MOOC_Syllabus_template.pdf
PDS-MOOC_Syllabus.pdf
PerTDL.c
PerTDL.o
untitled.cproj
week1-pres1.pptx
week1-pres2.pptx
week1-pres3.pptx
week2-lec1.mp4
week2-pres1.pptx
week2-pres2.pptx
week2-pres3.pptx
week2-rec1.cproj
week2-lec1.mp4
week2-lec2.mp4
week3-pres1.pptx
week3-pres2.pptx
week3-rec1.cproj
week3-rec2.cproj
week4-pres1.pptx
week4-rec1.cproj
TwoMoc
In-to-post
In-to-postfix.c
Intro-vid.mp4
ocw-guidelines.docx
pointers11.txt
pointers.txt
recursion11.txt
recursion.txt
stack-array.c
stack-array.o
stack-interface.h
stack-list.c
stack-list.o
stack.h
week1-lec1.mp4
week1-lec2.mp4
week1-lec3.mp4
week1-prog1.c
week1-prog2
week1-prog2.c
week1-prog3
week1-rec1.cproj
week1-rec3.cproj
week1-lec1.mp4
week2-lec2.mp4
week2-rec1.cproj
week2-rec2.cproj
$Narayanawamy-MacBook-Air:PS5-moc narayanawamy$ vi In
In-to-post In-to-postfix.c Intro-vid.mp4
$Narayanawamy-MacBook-Air:PS5-moc narayanawamy$ vi In
In-to-post In-to-postfix.c Intro-vid.mp4
$Narayanawamy-MacBook-Air:PS5-moc narayanawamy$ vi In-to-postfix.c

```

This called in-to-postfix dot c.

(Refer Slide Time: 18:40)

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include "stack-interface.h"

int isOperator(char c)
{
    return c == '*' || c == '+' || c == '/' || c == '-' || c == '^';
    // an operator can be any of the above characters
}

int precedence(char c)
{
    switch(c)
    {
        case '+':
        case '-':
            return 1; // + and - have precedence 1
        case '*':
        case '/':
            return 2; // * and / have precedence 2
        case '^':
            return 3; // ^ has precedence 3
        default:
            return -1; // this is to handle '(' case
    }
}

char * infixToPostfix(char exp[], int size)
{
    Stack *operators = createStack(size);
    char *output = (char *) malloc(sizeof(char)*(size+1)); // output to store the resulting postfix expres
    sion
    "In-to-postfix.c" 107L, 3184C

```

So, the first one is to check if the character is an operator or not. This is a very crucial thing and these are the operators; otherwise, we assume that, it is an operand. Then we also have the precedence rules. So, what is the precedence of C? So, we use this switch

function if plus and minus – you see they have the least precedence. So, we give them the precedence, which is 1. Star and division are seen; they give the precedence 2. And if the exponent is seen, it has the highest precedence; it is of precedence 3; otherwise, the precedence is minus 1; this is to handle the open parenthesis case.

(Refer Slide Time: 19:38)

```

    case '+':
    case '-':
        return 1;
    case '*':
    case '/':
        return 2;
    case '^':
        return 3;
    default:
        return -1;
}

// exp has the expression in infix w/
char * infixToPostfix(char exp[], int size)
{
    Stack operators = createStack(size);
    char *output = (char *) malloc(sizeof(char)*(size+1)); // output to store the resulting postfix expression
    int i = 0, k = 0;
    while(i < size)
    {
        if(isOperator(exp[i]))
        {
            while(!isEmpty(operators) && precedence(peek(operators)) >= precedence(exp[i]))
            {
                // pop until the precedence of the operator on top of stack is less than
                // precedence of the current operator
                output[k++] = peek(operators);
                pop(operators);
            }
            push(operators, exp[i]);
        }
        else if(exp[i] == '(')
        {
            push(operators, exp[i]);
        }
        else if(exp[i] == ')')
        {
            while(!isEmpty(operators) && peek(operators) != '(')
            {
                output[k++] = peek(operators);
                pop(operators);
            }
            pop(operators);
        }
        i++;
    }
    output[k] = '\0';
    return output;
}

```

So, now, here is the infix to postfix convertor. Sorry for the cleaning up operation; this is just to make the code more readable. So, what do I do? Expression; and recall that, the expression is an infix. So, exp has the expression in infix. And the number of symbols is size. So, now, you create a stack of size. Observe that, we actually do not use the whole stack when we create a stack, which is of same size of the expression. We actually are creating a stack, which is slightly larger. So, what do we do now? So, we create a stack of that particular size; recall that, we are going to use array implementation of the stack. So, therefore, we will get an array, which stores this whole expression. And here is the output, which is going to be a character string and it is going to have size plus one location – one additional location to store the null character.

So, here are the rules. If the symbol that is read as an expression; so while the operators are not empty and the precedence of the operator that you see at the top of the stack is greater than or equal to the precedence of expression of i; that is, as long as you see

operators of precedence, which are larger; you keep popping. And where does it go? Whenever you pop, it goes into the output. It is go into output of k, which is initialized to 0. Then you increment the counter; then you pop again. And when you exit, what is the condition for exiting? You exit either when the stack becomes empty or the operator that you see at the top of the stack is of precedence, which is strictly smaller. At this point, you push your operator on to the stack; otherwise, you now check if it is an open parenthesis; in which case, you blindly just push it on the stack; that is, push on to the operator's stack. On the other hand, if it is a close parenthesis, now what you do is while the operator stack is not empty and you do not see an open parenthesis, you keep...

(Refer Slide Time: 22:24)

```

/* exp has the expression in Infix */
char * InfixToPostfix(char exp[], int size)
{
    Stack *operators = (Stack*)malloc(sizeof(Stack)*size);
    char *output = (char *) malloc(sizeof(char)*(size+1)); // output to store the resulting postfix expression
    int i = 0, k = 0;
    while(i < size)
    {
        if(!isOperator(exp[i])) // if the character is operator
        {
            while(!isEmpty(operators) && precedence(peek(operators)) >= precedence(exp[i]))
            {
                // pop until the precedence of the operator on top of stack is less than
                output[k++] = peek(operators); // precedence of the current operator
                pop(operators);
            }
            push(operators, exp[i]);
        }
        else if(exp[i] == '(') // if the character is '(' push it into stack
            push(operators, exp[i]);
        else if(exp[i] == ')') // if the character is ')' pop until we get '('
        {
            while(!isEmpty(operators) && peek(operators) != '(')
            {
                output[k++] = peek(operators); // write all popped operators to output
                pop(operators);
            }
            pop(operators); // pop off the '(' symbol from stack
        }
        else
            output[k++] = exp[i]; // if the character is operand just write to output
        i++;
    }
    while(!isEmpty(operators))
    {

```

You take the top of the stack operator to and then push it into the output stream. Observe that, I could have written actually well; pop is a function that has not return any value. So, we read the value; write it into this location and then we pop it off. Observe that, there is a bit of program designed here. We have designed it, so that pop does not return any value; whereas, peek returns the value, which is the top of the stack. So, you keep popping till you see the open parenthesis. And when you see the open parenthesis, you pop the open parenthesis on the stack; otherwise, what do you know? You know that, it is not an operator; you know it is not a parenthesis – open or close. Therefore, it must be an operand. Of course, you must be careful when you use this function. I mean if I give you

a meaningless symbol, your formula may not be evaluated. Now you go to the next symbol. You repeat this procedure till the whole expression has been processed; till i is less than size.

(Refer Slide Time: 24:01)

```
int i = 0, k = 0;
while(i < size)
{
    if(isOperator(exp[i]))
    {
        // if the character is operator
        while(!isEmpty/operators) && precedence(peek/operators) >= precedence(exp[i]) )
        {
            // pop until the precedence of the operator on top of stack is less than
            output[k++] = peek/operators; // precedence of the current operator
            pop/operators;
        }
        push/operators, exp[i];
    }
    else if(exp[i] == '(')
    {
        // if the character is '(' push it into stack
        push/operators, exp[i];
    }
    else if(exp[i] == ')')
    {
        // if the character is ')' pop until we get '('
        while(!isEmpty/operators) && peek/operators != '(')
        {
            output[k++] = peek/operators; // write all popped operators to output
            pop/operators;
        }
        pop/operators; // pop off the '(' symbol from stack
    }
    else
    {
        output[k++] = exp[i]; // if the character is operand just write to output
        i++;
    }
}
while(!isEmpty/operators)
{
    output[k++] = peek/operators; // pop and write all the remaining operators to
    pop/operators;
}
output[k] = '\0';
return output;
```

Now, what you do; while the stack is not empty, you just keep printing out the operators, which are at the top of the stack; that is, keep sending them to the output. Finally, you put a null symbol, so that output becomes a string. Now, you return out...

(Refer Slide Time: 24:22)

```
while(!isEmpty(operators) && peek(operators) != '(')
{
    output[k++] = peek(operators);    // write all popped operators to output
    pop(operators);
}
pop(operators);                      // pop off the '(' symbol from stack
}
else
    output[k++] = exp[i];             // if the character is operand just write to output
    i++;
}
while(!isEmpty(operators))
{
    output[k++] = peek(operators);    // pop and write all the remaining operators to output
    pop(operators);
}
output[k] = '\0';
return output;
}

int evaluatePostfix(char* exp, int size)
{
    Stack* result = createStack(size);
    int i = 0;
    while(i < size)
    {
        if (isdigit(exp[i]))          // if the character is a digit
            push(result, exp[i] - '0'); // push it into the result stack
        else                          // if the character is an operator
        {
            int val1 = peek(result);    // read the top two values from
            int val2 = peek(result);
            // perform the operation
            // push the result back to the stack
        }
        i++;
    }
    return result->data[0];
}
```

I also have another small function, which is to evaluate a postfix expression. The input is given as a character array of a certain number of elements. Now, we create a stack of a particular size. This stack is going to store the value. Observe that, this particular program contains two functions: one is that, it evaluates a postfix expression; the other one converts an infix to a postfix expression. So, now, we have a result stack; the stack has size number of elements in it; you initialize i to 0. And now, i is less than size. So, what do we do? So, we look at the expression; we look at the character that we see from the expression; we check if it is a digit. isdigit is a function, which is there in the limits dot h... It is the string or the c type dot h library header. We will look at it just after this lecture as to which library header has this. So, you check if the character is a digit. And if it is in your digit, you push it on to the stack. What do you push on to the stack? You push the corresponding number. So, you take this character; subtract it from the ASCII value of 0 and we push it there.

(Refer Slide Time: 26:00)

```
        push(result, exp[i] - '0');           // push it into the result stack
    }
    else                                     // if the character is an operator
    {
        int val1 = peek(result);             // read the top two values from
        stack                                // stack
        pop(result);
        int val2 = peek(result);
        pop(result);                         // pop then while reading
        switch (exp[i])                     // based on the current operator perform the operation
        {
            accordingly
            case '+': push(result, val2 + val1); break;
            case '-': push(result, val2 - val1); break;
            case '*': push(result, val2 * val1); break;
            case '/': push(result, val2 / val1); break;
            case '^': push(result, pow(val2, val1));
        }
        ++i;
    }
    return pop(result);
}

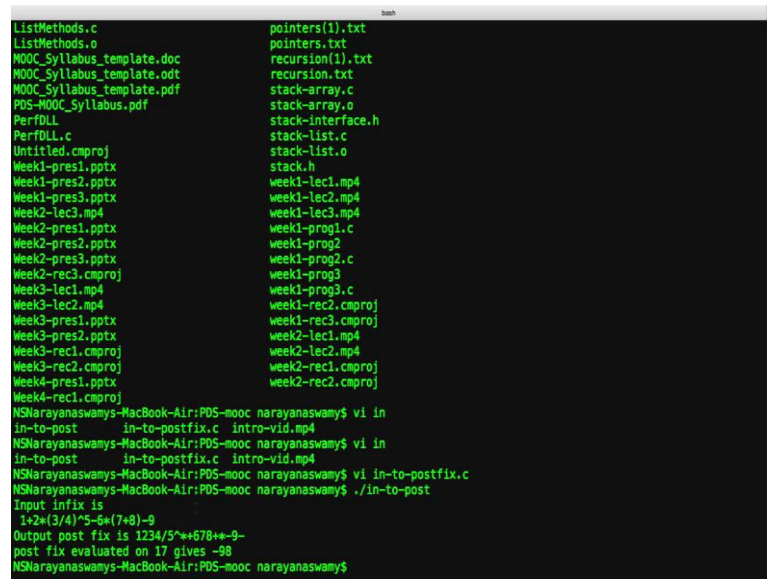
int main()
{
    char exp[50] = "1+2*(3/4)*5-6*(7+8)-9";
    char *output = infixToPostfix(exp, strlen(exp));

    printf("Input infix is %s\n", exp);
    printf("Output post fix is %s\n", output);
    printf("post fix evaluated on %d gives %d\n", strlen(output), evaluatePostfix(output, strlen(output)));
}
//in-to-postfix.c" 180L, 3212C written
```

Otherwise, you look at the top of the stack; that is the first argument. Then you pop it; then you look at the next top of the stack; that gives you the second argument and do this. And now, you do a switch. So, if you see that, the expression is... If you see that, this expression is a certain operator; so if it is plus; then you push on to the stack; the result, which is the sum of the two values; then you exit. If it is subtraction, you perform the subtraction; push it on to the stack. If it is multiplication, you perform the multiplication and push it on to the stack; and so on and so forth. And if it is exponentiation, we do the power function. This is the library function, which is in the math dot h library. Then we move on to the next symbol. At the end, the whole expression is been passed and we return the result.

This whole exercise is done on a single expression. So, let us just see this. The expression is 1 plus 2 multiplied by 3 by 4 with parentheses; the whole thing raise to the 5 minus 6 multiplied by 7 plus 8 minus 9. And then we make the appropriate functions. Output for example, contains the infix to postfix conversion. We use this string length function to give the next argument, which tells you the size of the stack; then we print out the infix expression; then we print out the postfix expression; and then we print out the answer after evaluating the expression on a certain value. In this case, the value is the length of the output itself.

(Refer Slide Time: 28:01)



```
listMethods.c      pointers(1).txt
listMethods.o      pointers.txt
MOOC_Syllabus_template.doc  recursion(1).txt
MOOC_Syllabus_template.odt  recursion.txt
MOOC_Syllabus_template.pdf  stack-array.c
PDS-MOOC_Syllabus.pdf      stack-array.o
PerfDLL             stack-interface.h
PerfDLL.c           stack-list.c
Untitled.cproj       stack-list.o
Week1-pres1.pptx     stack.h
Week1-pres2.pptx     week1-lec1.mp4
Week1-pres3.pptx     week1-lec2.mp4
Week2-lec3.mp4       week1-lec3.mp4
Week2-pres1.pptx     week1-prog1.c
Week2-pres2.pptx     week1-prog2
Week2-pres3.pptx     week1-prog2.c
Week2-rec3.cproj     week1-prog3
Week3-lec1.mp4       week1-prog3.c
Week3-lec2.mp4       week1-rec2.cproj
Week3-pres1.pptx     week1-rec3.cproj
Week3-pres2.pptx     week2-lec1.mp4
Week3-rec1.cproj     week2-lec2.mp4
Week3-rec2.cproj     week2-rec1.cproj
Week3-rec3.cproj     week2-rec2.cproj
Week4-pres1.pptx
Week4-rec1.cproj
NSNarayanawamys-MacBook-Air:PD5-mooc narayanawamys$ vi in
in-to-post          in-to-postfix.c  intro-vid.mp4
NSNarayanawamys-MacBook-Air:PD5-mooc narayanawamys$ vi in
in-to-post          in-to-postfix.c  intro-vid.mp4
NSNarayanawamys-MacBook-Air:PD5-mooc narayanawamys$ vi in-to-postfix.c
NSNarayanawamys-MacBook-Air:PD5-mooc narayanawamys$ ./in-to-post
Input infix is
1+2*(3/4)^5-6*(7+8)-9
Output post fix is 1234/5^~*678+-9-
post fix evaluated on 17 gives -98
NSNarayanawamys-MacBook-Air:PD5-mooc narayanawamys$
```

So, let us just run this program. So, the input in infix format is exactly what you had seen earlier. It is 1 plus 2 multiplied by 3 by 4 whole power 5 minus 6 multiplied by 7 plus 8 minus 9. Let us look at the output. So, the output in postfix form. So, let us just see if we can... So, 1 goes into the output stream; plus goes on to the stack; then 2 gets printed out; multiplication comes; multiplication has higher precedence on plus. So, this goes on to the top of the stack. Then the parenthesis goes on to the top of the stack; then 3 goes on to the top of the stack. Now, division is seen. So, division compared with open bracket, which is the top of the stack. So, division goes on to the top of the stack. Then 4 goes on to the top of the stack. Now, we see a close parenthesis; we print out the division, which is at the top of the stack; then we pop of the open bracket; that is, therefore, we print it out the division.

Then, we see exponentiation. Exponentiation has higher precedence than star. So, we push it to the top of the stack. And then 5 gets printed on to the output stream and so on. So, basically, you will be able to work this out by hand to check that, the output expression is indeed correct. And let us check that, the size of this array is actually 17. So, there are 1 to 9 are there. Then the number of symbols is that, between any two consecutive numbers, there is one additional symbol. Therefore... And there are two sets of parenthesis. So, you will actually see that, the size of this string is actually 17. The

size of the input expression is 17. And therefore, on evaluating it, you get minus 98. So, this is an easy exercise to work out by hand.

(Refer Slide Time: 30:35)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include "stack-interface.h"

int isOperator(char c)
{
    return c == '+' || c == '-' || c == '*' || c == '/' || c == '^';
    // an operator can be any of the above characters
}

int precedence(char c)
{
    switch(c)
    {
        case '+':
        case '-':
            return 1; // + and - have precedence 1
        case '*':
        case '/':
            return 2; // * and / have precedence 2
        case '^':
            return 3; // ^ has precedence 3
        default:
            return -1; // this is to handle '(' case
    }
}

/* exp has the expression in Infix */
char * infixToPostfix(char exp[], int size)
{
    Stack *operators = createStack(size);
    char *output = (char *) malloc(sizeof(char)*(size+1)); // output to store the resulting postfix expression
    infixToPostfix.c 188L, 3232C
```

So, let us just go back to the program for a minute. So, this is the function; it has two functions: one to convert infix to postfix; other one to evaluate a postfix expression. So, we have an operator function, we have a precedence checking function; then we have a conversion from infix to postfix, which carefully takes care of precedence of the operator. And then we evaluate a postfix expression on a given value.

So, that completes two more applications of the stack data type. You would have seen that, it is fairly sophisticated application; especially, the conversion from infix to postfix is a very very interesting application taking care of precedence and implementing the precedence rules by making careful checks with the values on the stack. The values on the stack remember – are operators. So, therefore, what we have done this week is to look at the stack data type. And we will give out one programming exercise, which involves using the stack data type in a couple of days time. So, hopefully, the lectures were enjoyable and the programming exercises will make you stronger. Keep programming and have fun.