Converting and Evaluating Infix, Postfix and Prefix Expressions in C

Rate Topic: 6 Votes

born2code
Posted 14 November 2007 - 02:43 AM

POPULAR

CONVERTING AND EVALUATING INFIX, POSTFIX AND PREFIX ExpressionS IN C
- Sanchit Karve

born2c0de
printf("I'm a %XR",195936478);

CONTACT ME : born2c0de AT dreamincode DOT net

CONTENTS

I. WHAT YOU NEED TO KNOW BEFORE YOU BEGIN

The reader is expected to have a basic knowledge of Stacks, Binary Trees and their implementations in C.
You will also need a C/C++ Compiler to run and test the source code.
All the source code in this tutorial has been tested on Visual C++ 6.0, but being standardized code it should work on any other C/C++ compiler.

The Algorithms given in this tutorial are not written as per any standard.
Yet (as you shall see soon enough) it is simple and easy to understand.
I have intentionally written it in a manner which is easier to understand with respect to my code.
Secondly, the example programs given below can be greatly optimized. But to make the code easier to understand, I had to compromise on optimization.
So please don't email/PM me about this.

All the C code given in this tutorial conform to C standards and is portable.

II. INTRODUCTION

Consider a situation where you are writing a programmable calculator. If the user types in 10 + 10, how would you compute a

result of the expression?
You might have a solution for this simple expression.
But what if he types in 3 * 2 / (5 + 45) % 10 ?
What would you do then?

There's no need to think about how you're going to compute the result because no matter what you do, it won't be the best algorithm for calculating expressions.
That is because there is a lot of overhead in computing the result for expressions which are expressed in this form; which results in a loss of efficiency.

The expression that you see above is known as Infix Notation. It is a convention which humans are familiar with and is easier to read. But for a computer, calculating the result from an expression in this form is difficult. Hence the need arises to find another suitable system for representing arithmetic expressions which can be easily processed by computing systems.
The Prefix and Postfix Notations make the evaluation procedure really simple.

But since we can't expect the user to type an expression in either prefix or postfix, we must convert the infix expression (specified by the user) into prefix or postfix before processing it.

This means that your program would have to have two separate functions, one to convert the infix expression to post or prefix and the second to compute the result from the converted expression.

This is what the tutorial is about. This tutorial will teach you the different techniques for converting infix expression to pre/postfix and then evaluate the converted expression to compute the result. You will be introduced to the conversion techniques first and later move on to writing the expression evaluator functions.

Once you read the tutorial, you will be ready to write your own programmable calculator and more.

Let us first introduce ourselves to the infix, postfix and prefix notations.

III. INFIX, POSTFIX AND PREFIX

Consider a simple expression : A + B
This notation is called Infix Notation.

A + B in Postfix notation is A B +

As you might have noticed, in this form the operator is placed after the operands (Hence the name 'post'). Postfix is also known as Reverse Polish Notation.
Similarly for Infix, the operator is placed INside the operands.

Likewise the equivalent expression in prefix would be + A B, where the operator precedes the operands.

So if X1 and X2 are two operands and OP is a given operator then
Quote
infix postfix prefix
X1 OP X2 = X1 X2 OP = OP X1 X2

We use infix notation for representing mathematical operations or for manually performing calculations, since it is easier to read and comprehend.
But for computers to perform quick calculations, they must work on prefix or postfix expressions. You'll soon find out why once you understand how expressions are evaluated.

Now let's take a slightly complicated expression : A + B / C
How do we convert this infix expression into postfix?
For this we need to use the BODMAS rule. (Remember?)
This rule states that each operator has its own priority and the operators with the highest priority are evaluated first. The operator priority in Descending order is BODMAS which is:

B O D M A S
Bracket Open -> Division -> Multiplication -> Addition -> Subtraction
[MAX. PRIORITY] [MIN. PRIORITY]

Hence, in the preceding example, B / C is evaluated first and the result is added to A.

To convert A + B / C into postfix, we convert one operation at a time. The operators with maximum priority are converted first followed by those which are placed lower in the order. Hence, A + B / C can be converted into postfix in just X steps.

:: A + B / C (First Convert B / C -> B C /)
1: A + B C / (Next Operation is A + (BC/) -> A BC/ +
2: A B C / + (Resultant Postfix Form)

The same procedure is to be applied to convert infix into prefix except that during conversion of a single operation, the operator is placed before the two operands. Let's take the same example and convert it to Prefix Notation.

:: A + B / C (First Convert B / C -> / B C)
1: A + / B C (Next Operation is A + (/BC) -> + A /BC +
2: + A / B C

Sometimes, an expression contains parenthesis like this: A + B * (C + D)
Parenthesis are used to force a given priority to the expression that it encloses. In this case, C+D is calculated first, then multiplied to B and then added to A. Without the parenthesis, B * C would have been evaluated first.
To convert an expression with paranthesis, we first convert all expressions that are enclosed within the simple brackets like this:
[INFIX TO POSTFIX]
:: A + B * (C + D)
1: A + B * C D +
2: A + B C D + *
3: A B C D + * +

Once an expression has been converted into postfix or prefix, there is no need to include the parenthesis since the priority of operations is already taken care of in the final expression.

IMPORTANT:
Keep in mind that converting this expression back to infix would result in the

same original infix expression without the paranthesis, which would ruin the result of the expression.
Only the Prefix and Postfix forms are capable of preserving the priority of operations and are hence used for evaluating expressions instead of infix.

IV. CONVERSION TECHNIQUES

Now that you know what infix, prefix and postfix operations are, let us see how we can programmatically convert expressions from one form into another.

This can be done by various methods but I'll be explaining the two most common techniques:
1) USING STACKS
2) USING Expression TREES

Let us study the first technique.

IV.A CONVERSION USING STACKS

I shall be demonstrating this technique to convert an infix expression to a postfix expression.
In this method, we read each character one by one from the infix expression and follow a certain set of steps. The Stack is used to hold only the operators of the expression. This is required to ensure that the priority of operators is taken into consideration during conversion.

Before you take a look at the code, read the Algorithm given below so that you can concentrate on the technique rather than the C code. It will be much easier to comprehend the code once you know what it is supposed to do.

The algorithm below does not follow any specific standard

Here's the Algorithm to the Infix to Prefix Convertor Function:

```
Algorithm infix2postfix(infix exp<b></b>ression string, postfix exp<b></b>ression string)
{
   char *i,*p;

   i = first element in infix exp<b></b>ression
   p = first element in postfix exp<b></b>ression

   while i is not null
   {
      while i is a space or tab
      {
         increment i to next character in infix exp<b></b>ression;
      }

      if( i is an operand )
      {
         p = i;
         increment i to next character in infix exp<b></b>ression;
         increment p to next character in postfix exp<b></b>ression;
      }

      if( i is '(' )
      {
         stack_push(i);
         increment i to next character in infix exp<b></b>ression;
```

Try converting an infix expression A + B / C to postfix on paper using the above algorithm and check if you're getting the correct result.

Now you can see how the above algorithm is implemented in C. In the following C code, I have added a 'whitespace adder' feature to the infix2postfix() function.
If the third parameter is 0, A + B / C will be converted as ABC/+
If the third parameter is 1, A + B / C will be converted as A B C / +

This doesn't seem like a big deal, does it?
But in actual practice we will be dealing with numbers as well. Which means that
32 + 23 would be converted to 3223+. This could mean 3 + 223, 32 + 23 or 322 +3.
Hence to prevent such cases, pass 1 as the third parameter to get 32 23 +.

Here's the Code:

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAX 10
#define EMPTY -1

struct stack
{
    char data[MAX];
    int top;
};

int isempty(struct stack *s)
{
    return (s->top == EMPTY) ? 1 : 0;
}

void emptystack(struct stack* s)
{
    s->top=EMPTY;
}

void push(struct stack* s,int item)
{
    if(s->top == (MAX-1))
    {
        printf("\nSTACK FULL");
    }
    else
    {
        ++s->top;
        s->data[s->top]=item;
```

Now try writing a program to convert infix to prefix as an exercise.

IV.B CONVERSION USING Expression TREES

Expression Trees are a slight variation of a Binary Tree in which every node in an expression tree is an element of an expression. It is created in such a way that an inorder traversal would result in an infix expression, preorder in prefix and postorder in postfix.

The main advantages of an expression tree over the previous technique are pretty obvious, easier evaluation procedures and efficiency.
The evaluation procedure of both the conversion techniques will be discussed in the next section.

The previous technique gives the resultant postfix expression as a string. If we now wish to convert the resultant postfix expression into prefix, we must pass the entire string to another (long) function which would provide the result into another string. If at any given point we wish to have all three forms of an expression, we would need three strings (of outrageously long lengths for longer expressions) whereas if we use an Expression Tree, we don't need anything else. We can choose which expression we want by choosing the appropriate

traversal method.
In short, unlike a string an Expression Tree can be interpreted as either infix,
postfix or prefix without changing the structure of the tree.

Consider an Infix expression : A + B / C
This expression would be represented in an Expression Tree as follows:

Quote

```
{+} (ROOT)
| |
--- ---
| |
{A} {/}
| |
--- ---
| |
{B} {C}
```

Let us Traverse this Expression Tree in Inorder, Preorder and Postorder.
Inorder : A + B / C
Preorder : + A / B C
Postorder : A B C / +

As you can see, Inorder Traversal gives us the Infix expression, Preorder the
Prefix expression and Postorder gives the Postfix expression. Isn't this
amazing? We don't even need a (long and complicated) conversion technique!!

The only thing that we need to do is convert an expression string into an
Expression Tree. This procedure is somewhat complex and requires a Stack as well.

This time I will demonstrate this technique to convert a Postfix Expression to
an Expression Tree.
Here's the algorithm for converting a postfix expression string to an Expression
Tree.

```
Algorithm postfix2exptree(postfix string, root<i.e. ptr to root node of exp tree>)
{
   NODES newnode,op1,op2;

   p = first element in postfix exp<b></b>ression;
   while(p is not null)
   {
      while(p is a space or a tab)
      {
         increment p to next character in postfix exp<b></b>ression;
      }

      if( p is an operand )
      {
         newnode = ADDRESS OF A NEW NODE;
         newnode->element = p;
         newnode->left = NULL;
         newnode->right = NULL;
         stack_push(newnode);
      }
      else
      {
         op1 = stack_pop();
         op2 = stack_pop();
         newnode = ADDRESS OF A NEW NODE;
         newnode->element = p;
         newnode->left = op2;
```

After the function is finished with execution, the root pointer would point to the root node of the expression tree.
Once again I'd like you to create an expression tree of A B C * + using the above algorithm on paper to understand how it works.

And finally, the implementation in C:

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

#define MAX 10
#define EMPTY -1


struct node
{
    char element;
    struct node *left,*right;
};

struct stack
{
    struct node *data[MAX];
    int top;
};

int isempty(struct stack *s)
{
    return (s->top == EMPTY) ? 1 : 0;
}

void emptystack(struct stack* s)
{
    s->top=EMPTY;
}

void push(struct stack* s, struct node *item)
{
```

As you can see, once the expression tree is built, the expression can be converted to any of the three forms by using an appropriate traversal method. I hope you can now understand its true advantage.

In this case since we were only dealing with alphabets (as variables in the expression), we chose a node structure like this:

```c
struct node
{
    char element;
    struct node *left,*right;
};
```

But in practice, we will be dealing with numbers (which will be more than a character in length is stored as a set of characters). Hence we would need two extra variables in the structure to hold the operator and the number as well as another flag variable which states whether the node contains an element or an operator. Hence the structure would look like this:

```
struct node
{
     char kind;
    char op;
    int number; /* even float would do */
     struct node *left,*right;
};
```

In this structure, if the node is to store an operator, it would store it in the op variable. If the node is required to store a number, it would do so in the number variable. If the node contains an operator the kind variable would have the value 'O' (or anything you wish) and 'N' for a number.
Take care not to name the op variable as operator since this will lead to compilation errors if compiled on a C++ compiler. C++ compilers won't compile the code since operator is a keyword in the C++ Language.
Not many people (today) use Pure C Compilers to compile C code (you're probably using a C++ Compiler right now too), it's best to avoid renaming it as operator.

I have chosen a char data type since it consumes only 1 byte rather than 2/4 bytes for an int for 16/32-bit based programes. This would reduce the size of the structure by 3 bytes.

Since we're on the topic of efficiency, there is another improvement that we can make to the structure definition.

Each Node in an expression tree can hold either an operator or a number, but not both. Hence if an operator is stored in a node, 2/4 bytes of space is wasted for the unused 'number' variable. If a node holds a number, there is a wastage of 1 byte for the unused op variable. How can we improve upon this structure design?

The answer is Unions. Unions are user-defined data types that can contain only one data element at a given time. The members of a union represent the kinds of data the union can contain. An object of union type requires enough storage to represent each member and hence its size is the same size of the largest member in its member-list.

This is what the improved structure definition looks like:

```
struct node
{
    char kind;
    union element
    {
        char op;
        int number;
    };
    struct node *left, *right;
};
```

This is an ideal situation for using a union.

As an exercise, try to write a function that builds an

expression tree for infix
and prefix expression strings using the node
structure that uses unions.

V. EVALUATION TECHNIQUES

The Evaluation process is the easiest part of an expression evaluator. Unlike the conversion process, evaluation functions are relatively small and easy to understand (and even write)

This is because now, we don't have to worry about the priority of operations anymore. Remember that evaluation of an expression is always performed on prefix or postfix expressions and never infix (this should be obvious by now).

As usual, I'll be dividing this section into two sub-parts.
V.A : EVALUATING Expression STRINGS
V.B : EVALUATING Expression TREES

V.A EVALUATING Expression STRINGS

It is fairly simple to evaluate postfix/prefix expression strings. The algorithm for evaluating a postfix expression is given below:

```
Algorithm evaluate(postfix exp<b></b>ression string)
{
   while current character in postfix string is not null
   {
      x = next character in postfix string;

      if( x is an operand )
         stack_push(x);
      else
      {
         op1 = stack_pop();
         op2 = stack_pop();
         result = op2 <operator> op1;
         stack_push(result);
      }
   }

   result = stack_pop();
   return result;
}
```

Simple isn't it? Let us see it in action:

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

#define MAX 50
#define EMPTY -1

struct stack
{
    int data[MAX];
    int top;
};

void emptystack(struct stack* s)
{
    s->top = EMPTY;
}

void push(struct stack* s,int item)
{
    if(s->top == (MAX-1))
    {
        printf("\nSTACK FULL");
    }
    else
    {
        ++s->top;
        s->data[s->top]=item;
    }
}

int pop(struct stack* s)
{
    int ret=EMPTY;
    if(s->top == EMPTY)
```

Now try writing an evaluator function for prefix expressions.

V.B EVALUATING Expression TREES

Expression trees are usually evaluated using recursion. The Recursive Evaluator is extremely simple.
Here is the Algorithm:

```
Algorithm evaluatetree(Node x)
{
   if( x is an operator )
   {
      op1 = evaluatetree(x.left);
      op2 = evaluatetree(x.right);
      result = op1 <operator> op2;
    }
    else
       return result; /* x is a number */
}
```

That's It!!! And there's no need to write two separate functions for postfix and prefix expressions.

Here's the Algorithm implemented in C code:

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

#define MAX 10
#define EMPTY -1

struct node
{
    char kind;
    char op;
    int number;
    struct node *left,*right;
};

struct stack
{
    struct node *data[MAX];
    int top;
};

int isempty(struct stack *s)
{
    return (s->top == EMPTY) ? 1 : 0;
}

void emptystack(struct stack* s)
{
    s->top=EMPTY;
}

```

Since I can't ask you to write an prefix version of the evaluator, write the same program using unions in the node structure as an exercise.
(Thought you could get away with it this time eh?)

VI. IMPROVEMENTS AND APPLICATIONS

You should now have a basic idea about converting and evaluating arithmetic expressions in C. You can add a lot of improvements by further adding support for more operators and functions such as sin(), cos(), log() etc in expressions.
This is a little complicated but not as difficult as it may seem. Simply assign a token to each function (like a single character 'S' for sine) and test for the token during evaluation and perform the appropriate function.

The whole idea of this tutorial is to give you an understanding of how arithmetic expressions are calculated by computers and embedded systems.

Don't reinvent the wheel by writing your own expression evaluators while writing programs. Languages such as C/C++,Java,C# etc. have in-built or external libraries that contain expression evaluators. Infact, using these libraries is recommended as they are more efficient and are less likely to be bugged.

Casio Scientific Calculators and many more use such techniques to solve
arithmetic expressions.
If you own a Casio fx-XXX MS/ES Series Calculator, you can even see that a stack
is being internally used (I currently use a Casio fx-991 ES).
If you key in a long expression with loads of operators and parenthesis and
press the '=' key, you will get a "Stack Error" or "Stack Full" Error.

Compilers also compute expressions using these techniques too.
Almost every mathematical program makes use of this too (Eg. Mathematica and
MATLAB)

That brings us to the end of this tutorial.
I hope you enjoyed reading this tutorial.

I also hope that you solve all the exercises

VII. CONTACT ME

I had starting writing the tutorial ages back and had almost abandoned it. Later,
I hurriedly completed the tutorial so although I have read through the tutorial
and tested all the code, there are still likely to be spelling mistakes or typos.
The code is written in pure, portable C but is tested on on Visual C++ 6.0, so
if I haven't declared variables at the beginning of a function, do let me know.

Please contact me if you find a bug or a typo or even if you need to clarify a
doubt.
Suggestions and Comments are welcome.
Irrespective of whether it's a suggestion, comment or a mistake, don't hesitate
to contact me on born2c0de AT dreamincode DOT net
You can also post a Reply to the thread where you found this Tutorial.
This post has been edited by **JackOfAllTrades**: 19 November 2011 - 06:58 AM
Reason for edit:: Removed gets() and fflush(stdin)

Replies To: Converting and Evaluating Infix, Postfix and Prefix Expressions in C

de_tutor
Posted 11 September 2008 - 09:41 AM

Thank you very much.

gangsta
Posted 05 October 2008 - 07:44 AM

its a good code.and its very good to put spaces between operands and operators.But I think you ignored that you put spaces.because while evaluating postfix expression, it gives wrong results for such an expression = "10+5",it cannot read number 10 as "ten" it assumes that its 1 "one".How can you correct that?

born2code
Posted 05 December 2008 - 12:37 AM

I have assumed every number to be a single digit in this tutorial.
If you wish to take into account of multiple digit numbers, keep traversing through the expression and append each traversed character to a string until an operator or a null character is found.
The string would then contain the number. After that, you can use atoi() and use the integer for evaluation.

Guitarded
Posted 15 December 2008 - 12:52 PM

Great tutorial, thanks!

I've modified it a bit to read multiple digit numbers and doubles for my project. There's only one thing I can't figure out, how do you make it read negative numbers and floating point numbers?

vellalyn
Posted 02 August 2009 - 09:26 AM

[quote name='born2c0de' date='14 Nov, 2007 - 01:43 AM' post='278982']

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAX 10
#define EMPTY -1

struct stack
{
    char data[MAX];
    int top;
};

int isempty(struct stack *s)
{
    return (s->top == EMPTY) ? 1 : 0;
}

void emptystack(struct stack* s)
{
    s->top=EMPTY;
}

void push(struct stack* s,int item)
{
    if(s->top == (MAX-1))
    {
        printf("\nSTACK FULL");
    }
    else
    {
        ++s->top;
        s->data[s->top]=item;
```

how to print or display the result of this..??
example if you in put 4 + (5 * 6)

the postfix would be 4 5 6 * +

but how will you display the answer which is 34..??

jslarochelle
Posted 22 August 2009 - 09:38 AM

Nice article. A little biased toward the tree implementation though. In fact, you don't have to keep working with strings when using the Stack version of the evaluator. You can convert the elements of the expression into say ExpressionElement as you do the infix to postfix conversion. After that the evaluator itself can work

directly with array or list of ExpressionElement. This way the Stack avoids some of the inconvenient you mentioned. I have found that in practice choosing between the two implementations is mostly a matter of taste and I have used both.
In my code for this kind of software I use lookup tables to tokenize the initial string into WORD (variables), NUMBER (literals), OPERATOR and sometimes FUNCTION. Lookup tables make the code much simpler and fast.

JS

---

tatsukitchy
Posted 20 September 2009 - 08:30 AM

can you convert that codes in VB6.0??

---

j7x
Posted 29 September 2009 - 06:04 PM

The code is nice b2c. We did this at the college - converting infix to postfix. I don't understand why we need to check for empty/overflow/underflow of the stack, please explain. Anyways, take a look at my code.

```
//Converting Infix to Postfix using exp<b></b>ressions.

#include<stdio.h>
#include<conio.h>

char post_fix[20]; // This declaration can be local too.
int input_precede(char sym)
{
    switch(sym)
    {
        case '+ : return 1;
        case '-' : return 1;

        case '*': return 3;
        case '/' : return 3;
        case '%': return 3;

        case '^': return 6;
        case '$': return 6;

        case '(': return 9;
        case ')': return 0;

        default: return 8;
    }
}

int stack_precede(char sym)
{
    switch(sym)
```

I've used different functions for assigning precedence, think it's simpler to understand that way.

#www#
Posted 23 October 2009 - 11:13 PM

please can you help me modifiing it to read multiple digit numbers >>and numbers maybein dicimal or hex or binary

please i want help>>>i have tried much...but it didn't work.

Guest_yonten*
Posted 30 March 2010 - 03:09 AM

sir,
it's good that you have given all the details about infix, postfix and prefix notation with good example and i learned a lot from there but what i have to know is that which expression evaluates to the largest number. for eg.
a)the prefix expression +*-2357
b)the infix expression 2+3*5-7
c)the infix expression (2+5)*(5-7)
d)the postfix expresion 23+57-*

i think prefix expression is the largest number but i am doubt whether it is a right answer.Therefore, may i have a correct answer.

Guest_aLoneKei*
Posted 17 June 2010 - 10:36 PM

Thank you!
The Code Worked Very Well!

Guest_george*
Posted 12 November 2010 - 10:09 AM

So what if you want to generate the expression tree not from the Postfix but from Prefix expression?

thefuzzyone
Posted 25 January 2011 - 10:55 AM

Just wanted to say thanks for the tutorial. It was just what I was after!

vividexstance
Posted 03 February 2011 - 01:52 PM

I've been able to do the stack version in C++ and now I'm trying to do the expression tree from a postfix expression. My question is, when you create nodes in the conversion function, you use malloc(), so I changed that to newnode = new node;, but you don't deallocate the memory anywhere, so does the code you present have memory leaks?

EDIT: I just tested it by defining a destructor for the node struct and it's not called at all for my code, which is very similar to born2codes program except where I changed C code to C++ code. I then made a function that is basically the same thing as postorder traversal except I delete the node instead of printing it. I then call this function at the end of main, and the node destructor prints out correctly for each node in the tree.
This post has been edited by **vividexstance**: 03 February 2011 - 02:14 PM

- (2 Pages)
- ▾
- 1
- 2
- →

Related C++ Topics beta

"Infix-Postfix & Postfix-Infix" Codes Problem

Solving For A Variable In Postfix - Evaluate Postfix, Solve For A Variable

Infix To Pre/postfix - Get Noticed On How To

Put Parenthesis In The Program

Converting A Postfix Expression To Infix Expression With Paranthesis - Help Needed

Infix To Postfix Program - Needing Help

Infix To Postfix Conversion

Problem With Postfix

Need Prefix Conversion Code...urgent!

Dynamic Array/struct

Converting Postfix, Infix, Prefix And Binary Trees In One Program - Conversion.....