Dijkstra's Shunting Algorithm with STL and C++

2008-07-08 By: Gabor Bernat

http://www.devarticles.com/c/a/Cplusplus/Dijkstras-Shunting-Algorithm-with-STL-and-C-plus-plus/

http://en.wikipedia.org/wiki/Shunting_yard_algorithm

http://www.chris-j.co.uk/parsing.php

In 1993 Alex Stepanov wrote the STL library. The library redefined the way people code in C/C++. Using it gradually became synonymous with elegance and speed. Later, as STL fought its way into the standards, it became even more popular and the numbers of people who used it increased exponentially. In the first two parts of this article series I presented the basics of STL; in this third part I will show you more complex code. You'll also learn how to write an expression evaluator/compiler.

The structure of the article will be the following: first, I will approach the problem of evaluating an arithmetic expression, how it can be completed and how we can make it simple. Here I should discuss the advantages of Reverse Polish notation, because, as you will see, evaluating an expression is almost child's play with it.

In the second part of this article, I'll approach the implementation problems, how Dijkstra's algorithm works, and I'll also present some code samples. The code will be full of STL and even include some templates, so I hope you have practiced to develop your knowledge of STL after you finished reading the first two parts and also the article covering C++ templates (search for it, too).

To understand the blocks of code, the information in those articles will be just as crucial as air in order to breathe. If you haven't until now I advise you to suspend this article and read those first.

In the early days of programming the first issue faced by the designers was how to create/allow the programmer to write an expression which is pretty close to the real mathematical form and evaluate it. It was crucial to be pretty close to the mathematical form as learning a new syntax and thinking about complex expressions in such a different way could overly complicate the life of the programmer and scare people away from new technologies. This was never an option.

Let's think about the problem a little. We have an expression like this:

!(x+y)*36+48/26*exp(3 && 4)

But as you will see later on, a simple idea implemented in an efficient and elegant way can reduce this scary-looking problem. However, back then getting a correct machine output was one of the biggest breakthroughs. In fact, in recognition of this, the first main programming language was named FORTRAN (it stands for FORMULA TRANSLATOR).

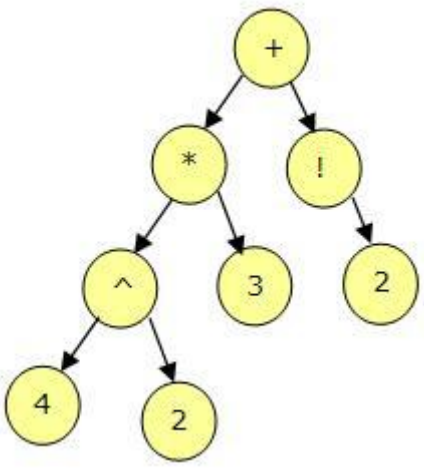
The main strength of the method that's going to be presented is that it doesn't need to do a repeated iteration through the expression, or to take account of the operator priorities or parentheses upon the evaluation. For this we'll first transform our input expression so that we'll only need to conduct iteration on the expression and find out the result. Let's start with the theory part.

The theory

Before looking deeper into the aforementioned Dijkstra's algorithm we should understand what we need to accomplish with it. And for this, we need to return to the binary trees. Yes, you read that right-binary trees. We are given the following expression:

!2+3*4^2

Now we're going to build from it an expression tree. An expression tree is a binary tree with the following main properties: the leaves are operands (numbers), while the interior vertices are operators. According to this we can create the following tree:



Now let's traverse the tree in a postfix way. This would result in the following expression:

2 ! 3 4 2 ^ * +

From here the expression can be evaluated effortlessly thanks to a genius Polish mathematician named JAN ŁUKASIEWICZ. In his memory the upper notation is named reverse Polish notation (**RPN**). Simplifying the problem to given numbers 3 and 4, Polish notation is called if we write first the operator followed by the operands, like this:

+ 3 4

And we talk about reverse Polish notation when there are first, the operands, and the operators are standing on the final positions. This is the same thing we have acquired after traversing the tree using the postfix form; it's just that we applied it recursively for more numbers/operators. Strictly on two numbers it looks like this:

3 4 +

If you still don't see how we can now solve the problem easily, let me put it in a different light. Take "a" and "b" numbers and the operator *. The evaluation can be made in different ways, depending upon which class we are currently using. We may say we have "a" and we need to multiply it with "b" on a math class.

Using Polish notation, we would say that we have operator * and we need to apply it to the following two numbers: "a" and "b". Now you can easily figure out that fort Reverse Polish Notation this would transform into a take number "a" and "b" and execute upon them collectively the * operator.

Of course, there are two types of operators: binary and unary. Binaries need to have arguments, however, a unary operator only needs one number. So for the unary operator we only need to take the number "b" into account. Apply the upper method for the RPN way of showing the result after Reverse Polish Notation and, once we finish the iteration through the expression the result should be at hand.

Calculating the input from the binary tree is reduced to iterating through RPN expressions -- when we find an operator, we execute it on the previous two (or one depending on the type of the operator) numbers. A step-by-step approach is below:

1) First item 2, no operator, put it in a stack.

2) Second item operator, unary. Get the last item from the stack and evaluate the operator on the number. Put the result back into the stack.

3) Third item number. Push it onto the stack.

4) Fourth item will be treated just as the previous one

5) Fifth item will be treated the same as the third and fourth items.

6) Sixth and the following are all binary operators, so we take the last two items form the stack. Evaluate them according to the operator and finally put the result back into the stack.

Following the stack's evaluation we shall see the following.

Stack:

1) 2

2) 0 // because !2 = 0

3) 0 3

4) 0 3 4

5) 0 3 4 2

6) 0 3 16 // because 4^2 = 16

7) 0 48 // because 3 * 16 = 48

8) 48 // because 0+ 48 = 48

What we have at the end in the stack is the outcome/result. If we have more than one item, we have an invalid input expression. Now that we know how to calculate the result, the question remains: how do we form, in a simple and an efficient way, Reverse Polish Notation? Here Edsger Dijkstra and his "Shunting algorithm" enter the picture. The name comes from the fact that the algorithm resembles that of a railroad shunting yard.

Getting into deep waters

We already observed from the previous section that if we build an expression tree with a simple postfix traversal of the tree, Reverse Polish Form is already acquired and solution isn't too far away. However, it looks as if building up the expression tree takes more resources and frustration that it should. Back in 1960 when Dijkstra himself faced this problem, he managed to figure out an optimal and also quite easy to behold algorithm.

The Shunting algorithm is a method for transforming mathematical expressions from the infix representation (what we write in our everyday life) to postfix/suffix/Reverse Polish form. The algorithm, just like the evaluation, is based on stacks.

The main idea is to get each token, item by item, from the input; if it is a number it will be added to the output stack. If it is an operator, which has a greater priority, we need to delay it until we find a lower one, and to accomplish this we must add the operator in another stack.

For implementing the algorithm I decided to create a class that will contain three data members: one for the infix form, one for the Polish form, and a final one that contains any error message which may occur. By choosing this path you can create, in the case of a compiler type expression, a sort of precompiled header since you may only need to check in those segments for mistakes where something additional was added.

To start, we need to define the operators and the applied priorities for them.

0) =

1) &&, ||

2) !

3) ==, != , <, > , <=, >=

4) +, -

5) *, /, %

6) ~ (replaces the - (negative)), ^

Additionally we'll note with -1 the priority for "(" and -2 the ")". For fast search of the priority we should use a map with the following declaration:

```
map<std::string,int> operator_Priority;
```

So with an operator_Priority["+"] for example we can find in minimal time the priority of the operator. But first let's see the code which manages the conversion part in order to understand more easily. First, declare a stack to hold the delayed operators and a string for the output, if we want to save the resulting expression for a later time and not just evaluate the expression. If you want only to evaluate the expression, then another stack should be enough. And we start iteration for each token from the input data.

```
stack<string> stackOperator;//delayed operator's stack

string token;



while( (type =nextToken(token)))

{
```

Now we act according to what type we have. First, the simpler ones: the parentheses. This is quite simple because whenever we find a "(" we just simply add it to the operator stack. Now as the "(" has a low priority, every other operator will go to the operator stack from now on -- at least until we get the closing match.

As soon as we have a closing parenthesis we start to go step-by-step through each item from the operator stack and push it to the end of the resulting expression. If, for whatever reason, we run out of operators on the operator stack and we still don't have the closing one, then we have an invalid data input that is lacking a ")" somewhere in it.

I have also implanted in the class a member for containing the error message for whatever reason the program fails to complete. By this I tried to bring this expression evaluator closer to the compilers as, upon failure, you can ask for the current error message from the class and display it on the screen so the user can make the needed

correction without being forced to observe for himself what the problem is in the input data. Additionally, the 0 type stands for invalid type as a result and invalid data entry.

```
switch(type)

{

case -1: // ( -> Push it on the delayed stack

{ stackOperator.push(token); break;}

case -2: // )

{

do {

if(stackOperator.empty()) {

error = "A unmatched ) operator has been found";

return false; }

topOperator = stackOperator.top();



if(topOperator != "(")

polishForm.push_back(topOperator); //add

stackOperator.pop(); //delete

}while(topOperator != "(" );

break;

}

case 0: // number

{

return false; break; } // error code
```

Now we have the issue of operators. Regardless of the type of the operator (unary or binary) the same steps should be covered, and due to the wisely created operator priority list we can kill two birds with one stone. So here it goes.

Moreover, unless we have an invalid operator we need to move to the output/polishForm all operators which are of lower priority until we find one which has a higher priority.

```
case 1: //unary operator

case 2: // binary operator -> cover them together

{

tokenPriority = priority(token);

if(tokenPriority == -3) // invalid operator

return false;

do

{

if (stackOperator.empty( ))

break;

else

{

topOperator = stackOperator.top();



iteratingPriority = priority(topOperator);

if(iteratingPriority == -1 || iteratingPriority < tokenPriority || iteratingPriority == 6)

break; //finish the iteration

else

{

polishForm.push_back(topOperator);
```

```
stackOperator.pop(); // remove the item from the stack }

}

}while(!stackOperator.empty());



stackOperator.push(token);

break;

}
```

If we have a number, simply add it to the polishForm. At the end, when we run out of tokens, we need to get each item from the stack and add it to the output.

```
case 3: //number

{

polishForm.push_back(token); //convert token and add

break;

}



}

}

while (!stackOperator.empty()) //what remained just add it

{

if(stackOperator.empty()) {

error = "A unmatched ) operator has been found";

return false;

}

topOperator = stackOperator.top(); //assign
```

```
polishForm.push_back(topOperator); //add

stackOperator.pop(); //delete

}
```

And that is it; with this little extra work we have created the code capable of making the transfer. Now apply the lesson learned on the previous page to this output/polishForm vector and the result will be at your disposal.

The class is written as a template so you can evaluate the expression just as easily with double, float, and even more. You may even use a specific class which handles large numbers or very precise ones if the specific operators for it are defined. If you are one of those really brave programmers I have also written an introduction to how to write yourself a class like this. It's a two-part series, and it will published soon here on ASP Free.

Epilogue

> And here is the code source. http://net.pku.edu.cn/~course/code/Dijkstra_Shunting/

In it you can observe much more interesting information which didn't make it into this article. Feel free to examine the information and try to understand each example. Needless to say, if you are having a hard time comprehending a specific expression please don't hesitate to post in the blog or take a step further and join the friendly, ever-growing forum DevHardware. After you have joined you can post your question, create a topic on it, and discuss your question with the entire community. An answer is guaranteed.

If you manage to assimilate each line of code from the source above, that clearly means that you have mastered the STL library. But if you want to get to that point you must apply yourself and research, ask questions, and ultimately get answers. Remember there aren't any wrong question, just wrong answers. If you want to get answers you need to ask questions.

Now that we are at the end of this three-part saga I hope you gained a good grasp of STL. You should be able to use it and understand how it works. As a bonus, whenever you need to write an expression evaluator or even a compiler for code you should already know from where to start and where to finish. Congratulations for sticking with me through all three parts and I wish you a great year!