Parenthesis Checking Using Stack in C Language

codewithharry.com/videos/data-structures-and-algorithms-in-hindi-33

In the last tutorial, we tried making parentheses matching intuitive and more understandable using stacks. We followed one simple algorithm to accomplish that.

The algorithm states:

- Everytime you come across an opening parenthesis, push it in the stack.
- Everytime you come across a closing parenthesis, pop one opening parenthesis out from the stack.
- We call this match of parentheses unbalanced when we encounter either of the two of these troubles:

1. There is no more opening bracket inside the stack to pop, and you come across a closing bracket.
2. The stack size is not zero, or there are still more than zero opening brackets present in the stack after you come across EOE(end-of-expression).

So, that was a quick revision of the things we learned in the previous tutorial. We did enough examples in the previous tutorial; you can check them as well. In today's lesson, we will program the algorithm in C.

Understanding the code snippet below:

1. Start by creating an integer function *paranthesisMatch,* and pass the reference to a character array(expression) *exp* in the function as a parameter. This function will return 1 if the parentheses are balanced and zero otherwise.

2. Inside that function, create a stack pointer *sp*. And initialize the size member to some big number, let it be 100. Initialize the top to -1, and assign the array pointer a memory location in the heap. You have the freedom to choose any data structure you want to implement this stack. We have learned stacks using both arrays and linked lists very efficiently.

```
struct stack* sp;
sp->size = 100;
sp->top = -1;
sp->arr = (char *)malloc(sp->size * sizeof(char));
```

Code Snippet 1: Creating and Initialising stack array.

3. So, it would be better if you just copy everything of stack implementation because it will more or less remain the same for that part. I'll use the array one.

4. Change the datatype of the array from integer to char. Accordingly, change everything from integer to char. And *arr* to exp.

5. Run a loop starting from the beginning of the expression till it reaches EOE.

6. If the current character of the expression is an opening parenthesis,'(' , push it into the stack using the push operation.

7. Else if the current character is a closing parenthesis ')', see if the stack is not empty, using isEmpty, and if it is, return 0 there itself, else pop the topmost character using pop operation.

8. In the end, if the stack becomes empty, return 1, else 0.

9. In the main, define a random character array expression and just passing this expression to *parenthesisMatch* would do our job.

Code for parentheses matching:

```
int parenthesisMatch(char * exp){
    // Create and initialize the stack
    struct stack* sp;
    sp->size = 100;
    sp->top = -1;
    sp->arr = (char *)malloc(sp->size * sizeof(char));


    for (int i = 0; exp[i]!='\0'; i++)
    {
        if(exp[i]=='('){
            push(sp, '(');
        }
        else if(exp[i]==')'){
            if(isEmpty(sp)){
                return 0;
            }
            pop(sp);
        }
    }

    if(isEmpty(sp)){
        return 1;
    }
    else{
        return 0;
    }
}
```

Code Snippet 2: Creating the parenthesisMatch function

Here is the whole source code:

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

struct stack
{
    int size;
    int top;
    char *arr;
};

int isEmpty(struct stack *ptr)
{
    if (ptr->top == -1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int isFull(struct stack *ptr)
{
    if (ptr->top == ptr->size - 1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

void push(struct stack* ptr, char val){
    if(isFull(ptr)){
        printf("Stack Overflow! Cannot push %d to the stack\n", val);
    }
    else{
        ptr->top++;
        ptr->arr[ptr->top] = val;
    }
}

char pop(struct stack* ptr){
    if(isEmpty(ptr)){
        printf("Stack Underflow! Cannot pop from the stack\n");
        return -1;
    }
    else{
        char val = ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}

int parenthesisMatch(char * exp){
    // Create and initialize the stack
    struct stack* sp;
    sp->size = 100;
    sp->top = -1;
    sp->arr = (char *)malloc(sp->size * sizeof(char));


    for (int i = 0; exp[i]!='\0'; i++)
    {
        if(exp[i]=='('){
            push(sp, '(');
        }
        else if(exp[i]==')'){
            if(isEmpty(sp)){
                return 0;
            }
            pop(sp);
        }
    }

    if(isEmpty(sp)){
        return 1;
    }
    else{
        return 0;
    }

}
int main()
{
    char * exp = "((8)(*--$$9))";
    // Check if stack is empty
    if(parenthesisMatch(exp)){
        printf("The parenthesis is matching");
    }
    else{
        printf("The parenthesis is not matching");
    }
    return 0;
}
```

Code Snippet 3: A program to check for balanced parentheses.

Let's now just see if the functions work properly. We will give it some expressions of our choice.

```
    char * exp = "((8)(*--$$9))";
    // Check if stack is empty
    if(parenthesisMatch(exp)){
        printf("The parenthesis is matching");
    }
    else{
        printf("The parenthesis is not matching");
    }
```

Code Snippet 4: Calling the parenthesisMatch function

The output we received was:

```
The parenthesis is matching
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output of the above program

Let's see for some another expression:

```
    char * exp = "8)*(9)";
    // Check if stack is empty
    if(parenthesisMatch(exp)){
        printf("The parenthesis is matching");
    }
    else{
        printf("The parenthesis is not matching");
    }
```

Code Snippet 5: Calling the parenthesisMatch function for another expression

The output we received was:

```
The parenthesis is not matching
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 2: Output of the above program

And if you could observe, this application of stacks uses almost all the operations we had learned before. From push and pop to isEmpty, everything. No doubt why this is the first one we are discussing.

Note: Parenthesis matching nowhere tells us if the given expression is mathematically valid or not. Because it is not supposed to, this algorithm has been meant just to return whether the parentheses in the expression are balanced or not.

For e.g., the expression ((8)(*9)) is mathematically invalid but has balanced parentheses.

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial, where we'll learn to match multiple parentheses using stacks. Till then, keep coding.