자료구조 HW#2

2016706018 박건희

**0.개발환경**



해당 과제의 모든 과정은 Visual Studio 2019에서 진행되었다.

**1.개요**

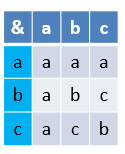
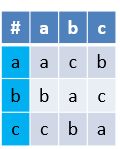
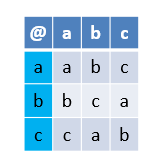
자료구조의 스택을 이용하여 계산기를 구현하였다. 이 계산기는 일반적인 계산기가 아닌 operand가 a, b, c인 symbol이고 binary operator가 @, #, &인 계산기이다. 이때, @, #보다 &가 연산자 우선순위가 높고, operation 결과는 operations.txt파일을 읽어서 해당 table대로 출력되도록 구현하였다. 이때. 입력을 개행(키보드의 엔터)으로 구분하고 요구사항대로 괄호간의 입력 오류만 존재한다고 가정하였다. 괄호간의 오류는 (,{,[와 ],},)가 매칭이 안되는 경우를 의미하고 각 괄호간에 우선순위는 동일하다.

**2.분석**

파일을 입력 받을 때, 예시인 operations.txt의 양식은 과제 설명을 따르면



위와 같고, 이 양식이 의미하는 바는



위 테이블과 같다.

그래서 파일을 입력 받을 때, @ # &의 table 순서는 달라지지 않으므로 개행 단위로 읽은 후 문자 a,b,c가 나오는 경우 순서대로 배열에 저장하도록 구현 가능하다. 데이터가 [3][3][3]으로 들어가는 3차원 배열에 입력 받은 Table을 저장하면 operand간의 연산시 해당하는 배열의 데이터에 접근하여 결과를 도출 가능하다.

operations.txt 입력에 관한 사항을 끝낸 후에 본격적인 계산기 구현을 위해 자료구조 stack을 사용하였다.

입력된 식의 문자 순서대로 Infix\_to\_postfix stack과 postfix stack에 각각 상황에 맞게 push,pop,연산을 진행하면 될 것이다.

이때, 경우를 나눠 볼 수 있다.

**(1). Infix\_to\_postfix stack (연산자 및 괄호 저장 stack)**

Infix\_to\_postfix stack에는 괄호”[ { (“과 연산자를 push한다.

Infix\_to\_postfix stack은 말 그대로 Infix로 입력받은 식을 postfix 식으로 바꿔주는 stack이다.

왼쪽 괄호”[{(”가 입력받은 경우 그냥 push하고,

연산자는 입력보다 Infix\_to\_postfix stack의 top이 연산자 우선순위가 낮거나 같은 경우, 연산을 진행한다.

연산 진행 과정은 연산 진행시, Infix\_to\_postfix stack에서 연산자를 pop하고 해당 연산자를 postfix stack에서 operand 2개를 pop하여 연산을 진행하면 될 것이다.

그 후 input을 Infix\_to\_postfix에 push한다.

위 과정은 Infix로 나타내진 식을 postfix로 바꿔주는 과정에 쓰이는 과정을 위한 stack이기 때문에 stack 이름을 Infix\_to\_postfix로 지정한다.

오른쪽 괄호 “] } )”가 입력으로 들어온 경우 연산을 진행한다. postfix stack에서 operand 2개를 pop하고 Infix\_to\_postfix stack에서 operator를 pop하여 연산을 진행 후 다시 push한다.

위 과정을 Infix\_to\_postfix stack에서 왼쪽 괄호 “[ { (”가 입력으로 들어올 때 까지 진행한다.

**(2).Postfix stack (operand 저장 스택)**

Postfix stack에는 operand를 push한다.

이 때, 연산과정이 필요하면 postfix stack에서 operand 2개를 pop하여 Infix\_to\_postfix stack에서 operator를 pop하여 연산을 진행 후 결과 값을 다시 push한다.

연산은 2개의 operand와 operator로 operations.txt에 표기된 table의 결과값을 의미한다.

위 과정이 마치 postfix로 나열된 식이 stack에서 차례차례 operand와 operator로 연산되는 과정과 유사하여 postfix stack으로 stack 이름을 지정한다.

최종적으로 postfix stack에 남게 되는 operand가 우리가 원하는 연산 결과 값이 될 것이다.

**최종 연산**

위 과정들을 들어오는 입력에서 반복하면서 연산하면 최종적으로 postfix에 operand가 남는게 이상적이지만 예외사항들이 존재한다.

입력을 전부 처리한 후에 연산되지 않은 operator가 stack에 존재하는 경우이다. 위 과정을 처리하기 위해 입력을 전부 처리한 후에는 남은 연산되지 않은 operator를 Infix\_to\_postfix stack의 size가 0이 될 때까지 연산을 진행한다.

**괄호 예외 사항**

위 과정을 따르면 연산이 가능하지만 괄호에 관한 예외사항들이 존재한다.

* **1. 연산 과정에서 열린괄호”[ { (”가 연산자 stack에 모자란 경우**

위 사항은 닫힌 괄호가 입력되어 열린 괄호가 pop될 때까지 연산할 때, stack을 전부 pop하여도 열린 괄호가 나오지 않는 경우를 의미한다. 그래서 위와 같은 사항이 발생하는 경우 error를 출력하도록 설정한다.

* **2. 최종 연산 과정에서 열린괄호”[ { (”가 연산자 stack에 남게된 경우**

위 사항은 최종 연산 과정에서 닫힌 괄호로 열린 괄호가 처리되어야 하지만 닫힌 괄호로 처리가 끝났음에도 불구하고 열린 괄호가 stack에 남아있는 경우이다. 이 경우도 괄호가 맞지 않는 경우이므로 error 출력 처리한다.

* **3. 소괄호, 중괄호, 대괄호 간의 매칭이 안되는 경우**

위 사항은 닫힌괄호로 인해 열린괄호와 매칭될 때까지 연산을 진행한 후 비교한다.

닫힌괄호가 대괄호이면 그에 해당하는 열린괄호도 대괄호가 pop이 되어야 하지만 이 부분이 맞지 않는 경우는 괄호의 매칭이 잘못된 경우이므로 error 출력 처리한다.

중괄호 소과호도 마찬가지로 해당하는 괄호와 매칭이 안되는 경우 error 출력한다.

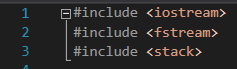
예를 들면 “**(a@b]**”는 괄호 자체는 서로 매칭이 되지만 왼쪽 괄호가 소괄호이고 오른쪽 괄호가 대괄호 이기 때문에 매칭이 되지 안아서 에러 출력 처리되어야 한다.

에러가 없이 정상적인 식으로 연산이 진행되었다면 최종적으로 Infix\_to\_postfix stack에는 아무것도 남지 않고, postfix stack에 최종 결과값이 push 되어 있을 것이다.

위 결과값을 출력하고 버퍼를 비운 뒤에 다시 입력받도록 구현하면 원하는 프로그램을 설계 가능하다. 또한 입력이 EOI인 경우에는 프로그램이 종료되고, 식에 관련없는 문자가 포함된 경우 무시하도록 설계한다.

**3. 구현**

위의 분석을 토대로 코드를 구현한다.

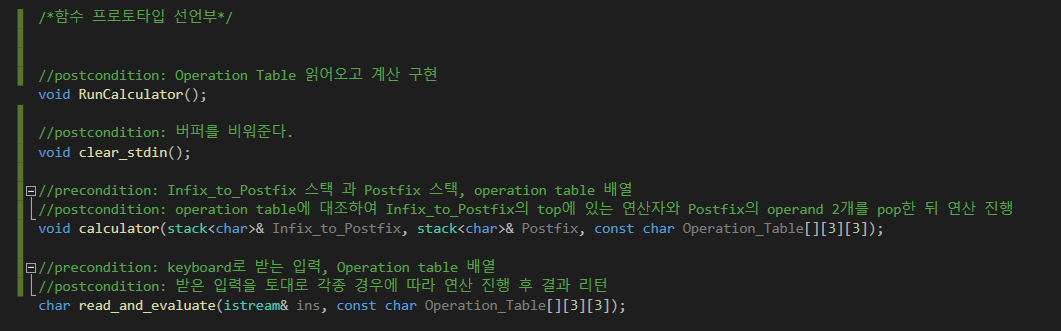


먼저 위와 같은 헤더파일을 사용한다.

입출력을 위한 **iostream**, 파일입력과 함수들을 사용하기 위한 **fstream**, stack 및 함수들을 사용하기 위한 **stack** 3개의 헤더파일을 사용하였다.

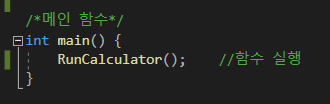


코드 구현시 번거로움을 줄이기 위해 namespace를 사용하였다.



함수는 4개를 제작하였다. 위 함수들의 precondition과 postcondtion은 명시한 것과 같다.

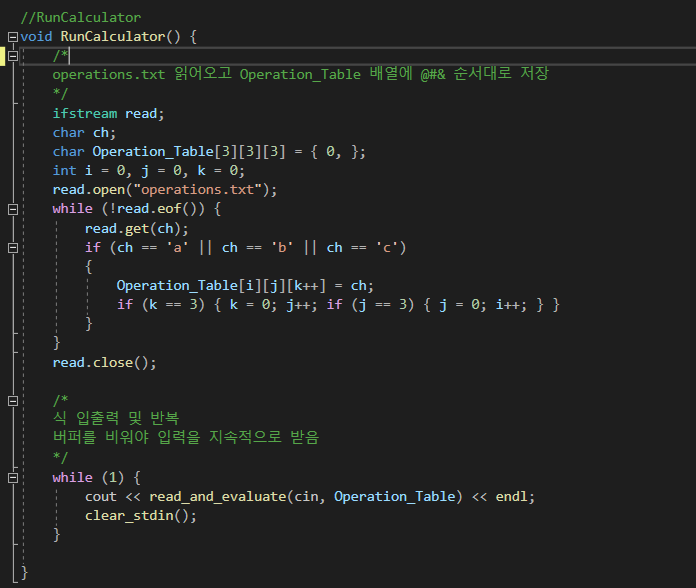
목표는 main함수에 다른 부가적인 코드 없이 함수만 실행하여 구현 가능하도록 하였다.



위와 같이 함수 하나로 코드가 실행되도록 하였다.

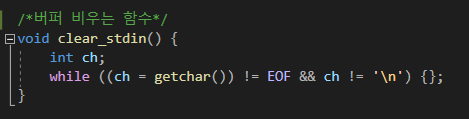
이제 함수들을 살펴보자.

**1. RunCalculator**



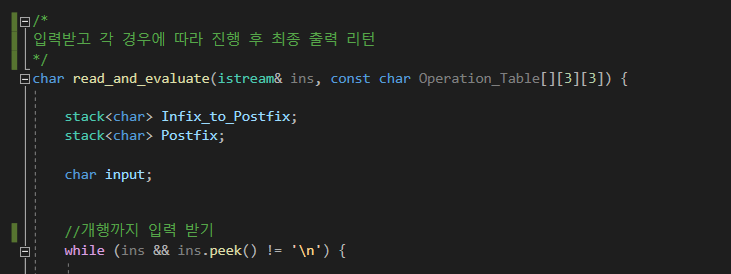
먼저 RunCalculator함수이다. main문에서 계산기를 실행해주는 함수이고, operations.txt를 읽은 후에 배열에 저장한다. 배열에 저장 후에는 파일을 닫고, 식 입력을 처리해준 후 결과값을 리턴하는 read\_and\_evaluate 함수를 무한루프로 진행한다. 이 때, 결과 출력후에 다시 함수 실행시 입력을 받지 않는 버그가 있어서 이를 막기위해 함수 결과 출력후에 clear\_stdin함수로 버퍼를 비워준 뒤 다시 입력을 받게된다.

**2. clear\_stdin**



버퍼를 비우는 과정은 위와 같이 버퍼에 문자가 비워질 때까지 루프 시켜 입력에 문제가 없도록 한다. getchar함수는 iostream 헤더파일에 포함된 함수이다.

**3. read\_and\_evaluate**



위 함수는 키보드 입력과 operation\_table을 입력 받는다. Operation\_table의 경우에는 위 함수상에서 사용하는건 아니고 연산과정에서 사용하지만 연산과정 함수(calculator)로 전달하기 위해 입력받았다.

위 함수에서 진행되는 과정은 식을 입력을 받고 결과를 리턴하는 과정이다.

핵심적인 부분들이 들어가서 각 경우를 처리해줘야 한다.

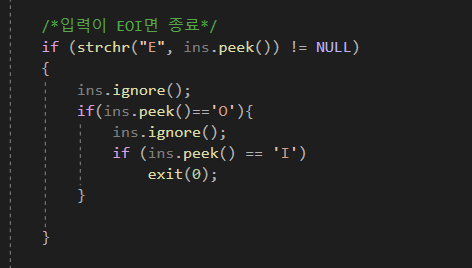
Stack은 2개를 사용하였는데, Infix\_to\_Postfix는 **2.분석**에 설명한 것과 같이 연산자와 왼쪽 괄호를 push하고 처리되는 stack이다. 의미로 언급한다면 Infix 식을 Postfix 식으로 변환해주는 과정에 사용되는 stack이다.

Postfix stack에는 operand가 저장된다.

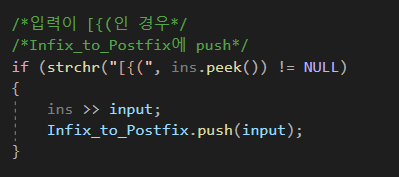
이 프로그램상에서는 postfix에 전부 나열 후에 처리하는게 아니라 연산이 필요할 때 즉시 진행되어 postfix stack에서 operand가 2개가 pop되어 연산 후에 다시 push된다.

while문을 통해 줄단위로 입력을 받는다. 입력된 식 문자열에서 문자열 순서대로 처리하게 된다.

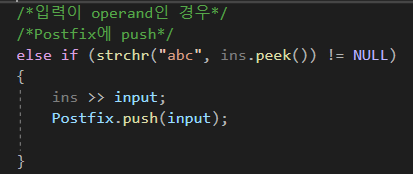
이 while문에서는 경우에 따라 결과를 처리한다.



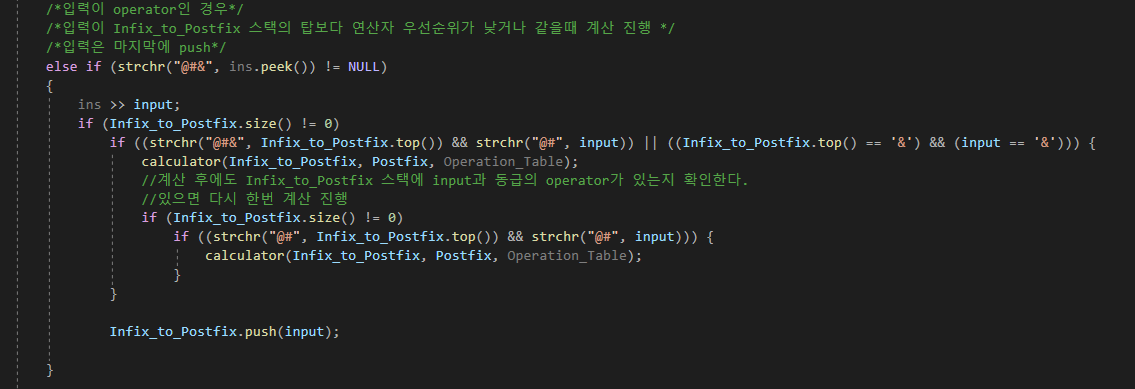
먼저 프로그램을 종료하는 EOI가 입력된 경우 exit(0)함수를 이용하여 프로그램을 종료한다.



입력이 왼쪽괄호 “[ { (”인 경우 분석에 명시한대로 Infix\_to\_Postfix에 push한다.



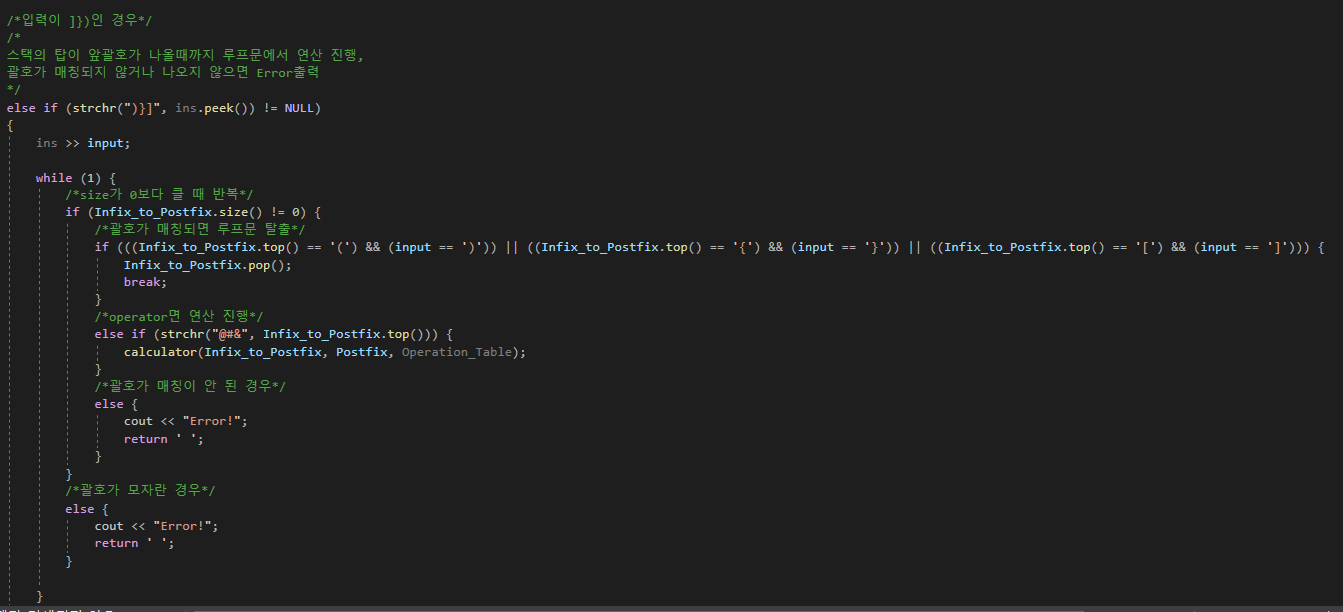
입력에 operand인 경우 Postfix에 push한다.



입력이 operator인 경우 Infix\_to\_Postfix stack의 top에 연산자가 존재할 때 연산자가 입력보다 우선순위가 낮거나 같은 경우에는 연산을 진행한다.

위와 같이 @#&의 입력에서 연산 진행 후에도 stack을 검사하여 stack의 top에 @,#이 존재하는 경우와 input이 연산자 우선순위가 동급인 @,#인 경우 연산을 다시 한번 진행한다.

연산을 진행하는 calculator 함수는 밑에서 설명하고, 위 과정이 진행되거나 진행할 필요가 없었더라도 Infix\_to\_Postfix stack에 마지막에 입력을 push한다.

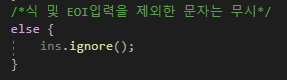


입력이 오른쪽괄호 “] } )”인 경우 여러가지 경우들을 고려해주어야 한다.

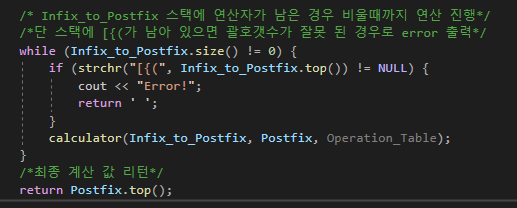
무한루프를 통해 Stack의 top이 연산자인 경우 연산과정을 진행하고,

오른쪽괄호가 매칭되는 왼쪽괄호가 pop이 될 때 루프문에서 빠져나오도록 구현하였다.

하지만 왼쪽괄호를 만나더라도 매칭이 안된경우가 존재하고, 최종 출력 과정에서 왼쪽 괄호가 안나오는 즉, 괄호가 모자란 경우가 존재하므로 위 2가지 예외의 경우에는 error가 출력된다.



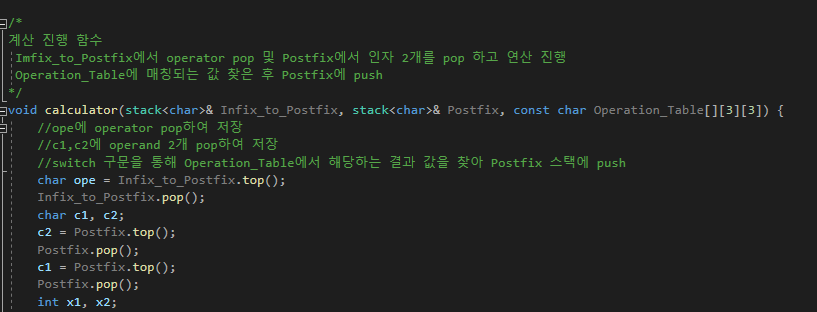
마지막으로 상관없는 문자가 식 사이에 들어왔을 때 무시해주도록 구현하였다.



위 과정들을 진행시, 입력을 전부 처리하더라도 operator가 stack에 남아있는 경우가 존재한다.

위 과정들을 처리하기 위해 stack의 size가 0이 될 때까지 연산을 진행한다. 하지만 만약 이 때, 왼쪽 괄호가 stack에 존재한다면 괄호가 매칭이 안된 경우이므로 error를 출력한다.

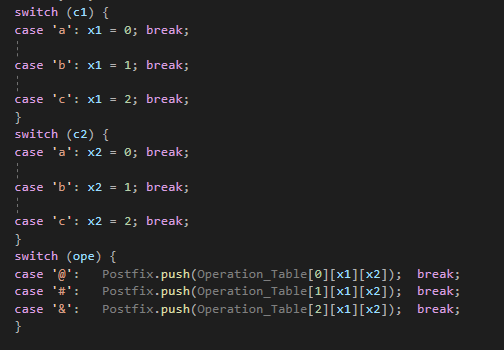
**4. calculator**



연산 진행시마다 호출되는 함수이다.

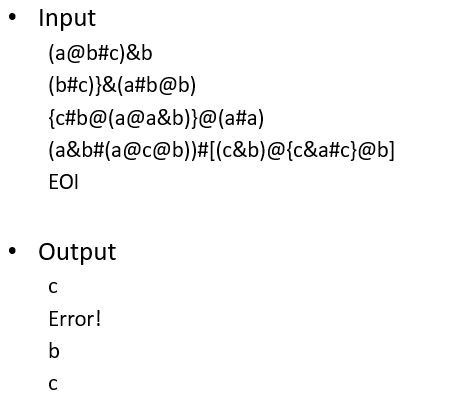
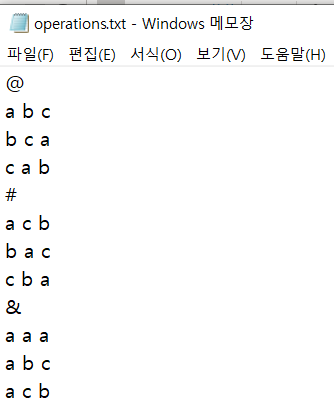
연산자를 스택에서 pop하고 해당 연산자와 operand를 스택에서 2개 pop하여 연산한다.

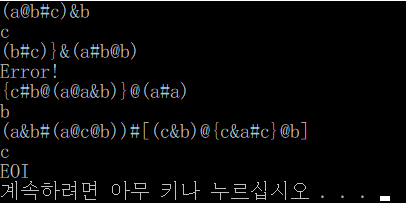
이제 operator와 operand 2개의 정보가 존재하므로 operation\_table에서 매칭되는 결과를 찾을 수 있다.



switch문을 통해 operand 2개의 경우에 따라 x1과 x2를 변화시키고, operator가 @,#,&냐에 따라 operation\_table에서 해당하는 결과를 찾아 postfix stack에 push한다.

**4. 테스트**

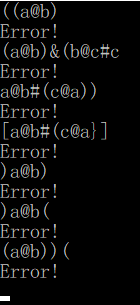
****



먼저 간단하게 예시 operations.txt를 통한 테스트이다.

결과가 ppt와 같이 정상적으로 출력된 것을 확인하였다.

그 다음 error 출력을 위해 괄호가 잘못 매칭되는 경우들을 input으로 넣어보았다.



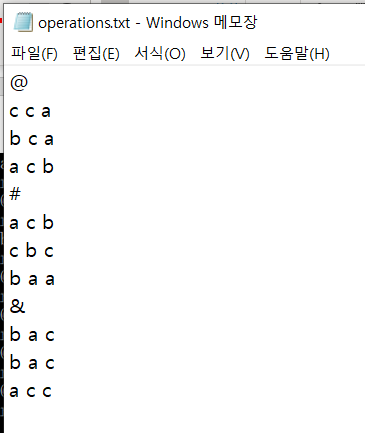
위와 같이 괄호가 잘못된 경우들은 모두 Error가 출력된다.



EOI 입력시 프로그램이 종료되는 것을 확인하였다.

그 이후 테스트는 operations.txt를 변형하였다.

**테이블 변경**



Table을 위와 같이 무작위로 바꾸어 위와 같은 상황에서 test를 진행하였다.

먼저 연산자 우선순위 테스트를 해보았다.



입력을 위와 같이 a@b&c를 줬을 때

우선순위가 지켜지지 않는다면 (a@b)&c가 되어 c가 출력되지만

&가 우선순위가 높으므로 a@(b&c)로 연산이 되고 결과는 a로 정상적으로 출력된 것을 확인하였다.



테스트를 위해 이번에는 복잡한 input을 넣어보았다.

위 테이블을 직접 table과 매칭시켜 풀어보면 결과가 a가 나오고 프로그램의 결과와 같기 때문에 정상적으로 작동한 것을 확인하였다.



좀 더 복잡한 input을 넣어도 실제 연산 후 비교했을 때 정상적으로 작동하는 것을 확인하였다.



마지막으로 input을 넣어서 테스트 해봤을때, 결과가 실제 연산과 같은 c로 정상적으로 작동하는 것을 확인하였다.

**5. 고찰**

stack을 통한 특수한 계산기를 구현하였다. 계산 과정은 스택을 이용해 Infix를 Postfix로 바꾸는 과정과 Postfix 스택에서 operation table을 바탕으로 연산이 진행되도록 하였다. 이 계산기에서 고려할 부분이 정말 많았다. 파일 입력시 불러오는 operations.txt의 형식이 어긋나면 제대로 된 연산이 불가능하기 때문에 정확한 양식대로 txt파일을 만들어야 했고, 스택이 계산기로 구현 가능한 이유를 정확히 고려해야했다. 또한 특수한 상황들이 발생하므로, 예를 들면 괄호가 잘못되거나 연산자 우선순위가 다르거나 그러한 부분들을 고려해야 했다.

Operations.txt를 입력 받는 경우를 살펴보면, 배열에 집어넣었는데 처음에 배열을 편의를 위해 전역변수로 선언을 했으나 추후 프로그램이 길어질 때 버그가 생기는 영향이 만에 하나라도 생길 수 있기 때문에 지역변수로 저장 후에 const 포인터로 전달하는게 더 좋다고 판단하였다.

3x3x3배열에 for문을 통해 @#&순서대로 테이블을 저장 후에 함수를 진행하는데 분석에서 가장 까다로웠던 부분은 연산자 또는 오른쪽 괄호가 입력되었을 경우이다.

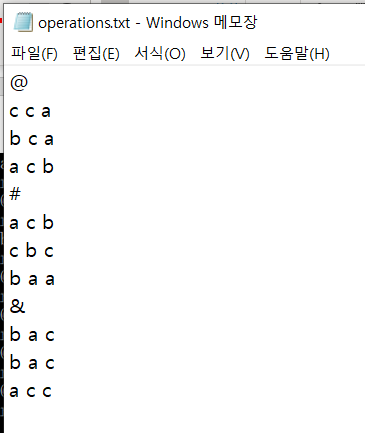
연산자의 경우는 스택의 사이즈가 0일때 스택의 top을 리턴시키면 강제종료 되기 때문에, 사이즈가 0인지 검사 후에 0이면 그냥 push하였다. 그 후 0이 아니면 스택의 탑이 왼쪽 괄호인지 또는 연산자일때는 우선순위를 고려하여야 했다.

오른쪽 괄호가 입력되면 경우를 따져야 했다.

스택에서 왼쪽 괄호가 나올 때까지 연산을 진행하였고 이때 대괄호 중괄호 소괄호 매칭이 안되거나 나오지 않는 경우는 에러를 출력했다.

그리고 입력을 전부 받았을때 연산자가 스택에 남은 경우 왼쪽 괄호가 남은건지 검사와 동시에 연산을 진행하였다.

작성한 코드를 기반으로 테스트시 발생한 버그 중에서 가장 난해했던 버그를 살펴보면





위 테이블을 기반으로 직접 풀어보았을 때와 식이 결과가 다른 버그를 찾아냈다.

직접 계산했을 때 답이 b가 나왔는데 프로그램 결과와 다르게 나온 것이다.

연산자 우선순위 부분이 문제가 된 것으로 추정된다.

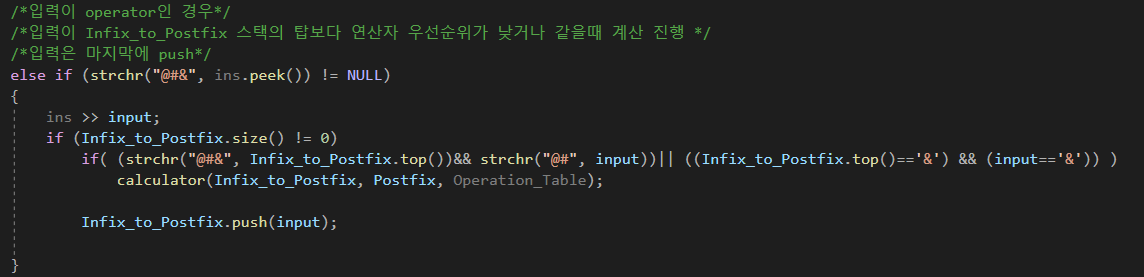
디버깅을 하면서 무엇이 문제인지 찾아보았다.



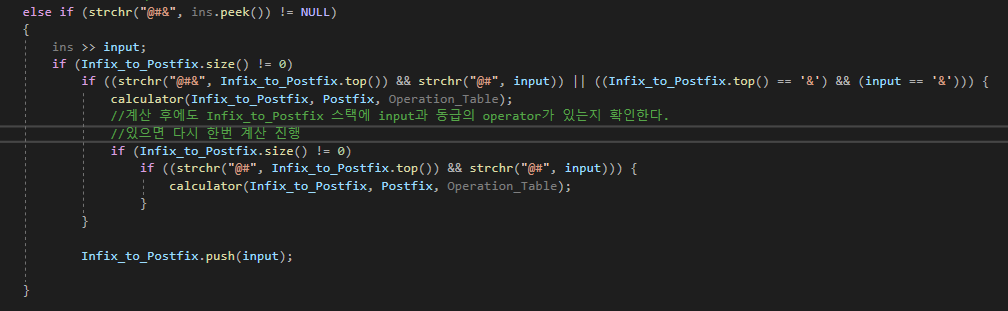
이 부분에서 버그가 일어나는데 이유는 위의 식은 테이블대로 진행했을 때

c&a는 a인데 &연산 과정이 입력이 @가 들어왔을 때 진행되는데 &연산 후에 &가 pop되고 나서 @가 stack에 push된다. 즉 stack에는 @가 2번 push된 상태이고, 그 후 b@a@c에서 a@c가 먼저 계산되어 b@(a@c)와 같이 식이 계산된다. 우리가 원한 순서는 (b@a)@c이고 이 문제는 operations table이 결합법칙을 만족하지 않는 table이면 발생하게 된다. 이 문제를 해결하기 위해 연산 후에 stack의 top에 @나 # 연산자가 있는 경우, 입력이 @나 #이면 다시 한번 연산을 진행하도록 수정하였다.

**기존 코드**



**변경 후 코드**



버그 수정을 위해 위와 같이 @#&의 입력에서 연산 진행 후에도 stack을 검사하여 stack의 top에 @,#이 존재하는 경우와 input이 연산자 우선순위가 동급인 경우 연산을 다시 한번 진행한다.

그 결과,



버그가 수정된 것을 확인하였다.

이외에도 자잘한 버그들이 정말 많았으나 경우를 따져가며 fix하였고 필요한 함수들은 구글링을 통해 해결하였다.