REPORT

과목	데이터 구조
담당교수	김인겸 교수님
학과	정보통신공학
학번	20190895
이름	김찬영

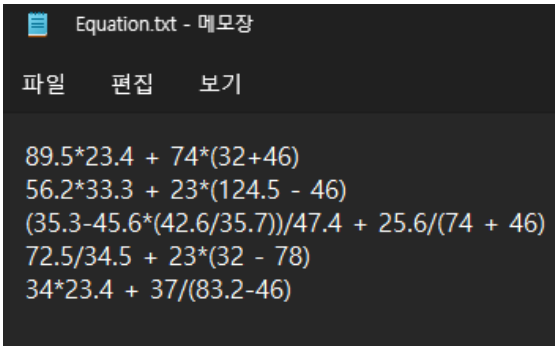
Contents

1. 과제 목표 분석

이번 과제의 목표를 분석해보면, 다음의 조건을 만족하는 전자계산기를 프로그래밍 하는 것이다.

1) 수식이 저장된 Equation.txt 파일이 있음.
2) Main()함수의 Arguments를 이용하여 파일을 읽은 후 check 값에 따라 큐 또는 스택을 이용하여 처리한다.
3) 위의 수식을 순서대로 스택(또는 큐)에 저장한 후 pop()(또는 dequeue())함수를 이용하여 각각의 수식을 꺼내 계산한 결과 값을 보여준다.
4) 큐와 스택은 연결리스트를 이용하는 것을 원칙으로 하나, 배열로 처리해도 좋음.
5) Equation.txt의 수식이 스택에 들어갔다가 나오면서 차례대로 결과를 보여줌.

아래는 Equation.txt의 내용과 명령어를 포함한 계산기의 결과 화면을 보여준다.



Calculator Equation.txt 1

Result = 796.594624
Result = -1055.898551
Result = -0.189904
Result = 3676.960000
Result = 7866.300000

2. 스택을 이용한 전자계산기 구현

과제 해결을 위해, 문제점들을 우선 생각하였다.

1) 가장 먼저, Equation.txt를 읽을 때와 괄호 검사, 수식 계산 등 계산기를 실행할 때 스택이 둘 다 필요하다. 또한, 스택은 연결리스트로 구현해야 한다.

2) 이전 과제에서 프로그래밍한 전자계산기는 한 자리의 숫자밖에 사용하지 못하였는데, 그 이유는 문자열을 스캔할 때 한 글자씩 스캔하고 스캔한 글자에 따라 조건문을 사용하는 방식이었기 때문이다. 두 자리 숫자나 소수를 스캔하는 방법에 대해서 생각해야 한다.

위의 두 가지 문제만 해결된다면, 이전에 프로그래밍했던 스택을 사용한 전자계산기와 크게 다르지 않기 때문에 쉽게 프로그래밍 할 수 있다.

우선 1번 문제의 해결은, 간단하게 스택을 2개 구현함으로써 해결할 수 있었다. 원래의 스택은 double형 변수를 저장하게 선언하고, 계산기 동작에 관여하는 스택으로 사용한다. 기존의 스택 알고리즘과 동일하다. 2번째 스택은 문자열을 저장하게 선언하고, 그에 맞는 스택 연산 함수들을 선언한 후 텍스트 파일로부터 한 문장씩 읽어들일 때 순서대로 스택에 저장하는 방식으로 사용한다.

(1) 계산기의 여러 함수들 및 문자열의 입력을 위한 스택의 구현

```
//에러 출력 함수
void error(char* str)
{
	fprintf(stderr, "%s\n", str);
	exit(1);
}


//연결리스트를 이용한 스택
typedef double Element;
typedef struct LinkedNode {
	Element data;
	struct LinkedNode* link;
} Node;
Node* top = NULL;
//스택이 비어있는지 확인하는 함수
int is_empty() { return top == NULL; }
//스택 초기화 함수
void init_stack() { top = NULL; }
//스택 안의 요소의 개수를 반환하는 함수
int size(){
	Node* p;
	int count = 0;
	for (p = top; p != NULL; p = p->link) count++;
	return count;
}
//스택 삽입 연산
void push(Element e){
	Node* p = (Node*)malloc(sizeof(Node));
	p->data = e;
	p->link = top;
```

```
        top = p;
}
//스택 삭제연산
Element pop(){
        Node* p;
        Element e;
        if (is_empty())
                error("스택 공백 에러");
        p = top;
        top = p->link;
        e = p->data;
        free(p);
        return e;
}
//스택 peek연산
Element peek(){
        if (is_empty())
                error("스택 공백 에러");
        return top->data;
}
```

```
/*문자열을 저장하기 위해 선언한 연결리스트를 이용한 스택2
기존의 스택 연산과 동일하며 노드 안에 문자열을 저장할 수 있게 하였다*/
typedef struct LinkedNode2 {
        char* str;
        struct LinkedNode2* link2;
}Node2;
Node2* top2 = NULL;
int is_emptystr() { return top2 == NULL; }
void init_stackstr() { top2 = NULL; }
void pushstr(char* expr) {
        Node2* p = (Node2*)malloc(sizeof(Node2));
        p->str = expr;
        p->link2 = top2;
        top2 = p;
}
char* popstr() {
        Node2* p;
        char* str;
        if (is_emptystr())
                error("스택 공백 에러");
        p = top2;
        top2 = p->link2;
        str = p->str;
        free(p);
        return str;
}
```

2번 문제의 해결하기 위해 두 가지를 생각했다. 우선, 중위 표기식을 후위 표기식으로 변환해야 하고, 변환된 후위 표기식을 계산해야 하는데, 후위 표기식으로 변환만 할 수 있으면 계산하는 프로그램을 코딩하는 것은 어렵지 않다고 생각했다. 그럼 먼저 중위 표기식을 후위 표기식으로 변환하는 함수 infix_to_postfix()에 대해서 설명하겠다.

(2) `void infix_to_postfix(char* expr, char* postexpr)`

예를 들어, 32.14-5라는 수식이 있다고 생각해보자. 먼저 이 식을 후위 표기식으로 바꾸기 위해서는, 32.14와 -와 5를 모두 독립적으로 받아야 한다. 연산자의 경우, 하나의 문자로만 구성되기 때문에 문제가 되지는 않지만 실수의 경우, 한 문자로 표시되는 것이 아닌 여러 문자열로 표시되기 때문에 이를 프로그램이 어디서부터 어디까지 하나의 실수인지를 인식시키는게 중요하다고 생각했다.

(1) 문자열을 한 문자씩 읽어들인다.

(2) 만약 읽어들인 문자가 숫자일 경우, 숫자나 점이 끝날 때 까지 배열에 읽어들인다. 이 과정을 통해 실수를 입력받을 수 있고, 실수의 입력이 끝나면 배열에 공백을 추가해 한 실수의 입력이 끝났음을 구분한다.

(3) 또한 연산자를 읽었을 경우, 연산자 사이에도 공백을 추가하여 확실하게 실수와 실수, 실수와 연산자를 구분한다!

(4) 나머지 괄호 및 연산자의 우선순위는 이전의 스택의 전자계산기 알고리즘과 동일하다.

위 과정을 거쳐 완성한 함수는 다음 사진과 같다.

```c
/*중위 표기 수식을 후위 표기 수식으로 변환하는 함수
한 자리의 숫자가 아니기 때문에 실수와 실수, 실수와 연산자를 구분할 수 있도록 각각의 사이에 공백을 추가해 구분한다.*/
void infix_to_postfix(char* expr, char* postexpr) {
    init_stack();
    while (*expr != '\0') { //문자열의 끝을 만나기 전 까지 반복

        if (*expr >= '0' && *expr <= '9') { //조건문 : 실수를 읽기 위한 조건문. 숫자나 점을 만났을 경우
            do {
                *postexpr++ = *expr++;  //postexpr에 expr의 값을 대입하고 각각 포인터가 가리키는 위치 1 증가하면 다음 글자를 계속 읽을 수 있다
            } while ((*expr >= '0' && *expr <= '9') || *expr == '.');   //do-while문을 이용해 계속 postexpr에 대입한다
            *postexpr++ = ' ';  //실수를 다 읽으면 postexpr에 공백을 추가
        }

        else if (*expr == '(') {    //조건문 : 좌괄호를 만났을 경우
            push(*expr++);  //우선 스택에 저장하고 expr의 위치 증가
        }

        else if (*expr == ')') {    //조건문 : 우괄호를 만났을 경우
            while (peek() != '(') { //좌괄호를 만나기 전 까지(반복문 안에서 pop을 하기 때문에 peek을 사용해도 무한 반복되지 않는다)
                *postexpr++ = pop();    //스택에 저장된 연산자를 pop 해서 postexpr에 대입한다
                *postexpr++ = ' ';  //연산자를 pop 하면서 연산자 사이에 공백을 추가
            }
            pop();  //좌괄호를 스택에서 pop한다
            expr++; //괄호 안의 연산자를 다 꺼냈을 경우 expr 위치 증가
        }

        else if (*expr == '+' || *expr == '-' || *expr == '*' || *expr == '/') {    //조건문 : 연산자를 만났을 경우
            while (!is_empty() && (precedence(*expr) <= precedence(peek()))) {  //스택이 비어있지 않고, expr의 연산자가 스택에 저장된 연산자보다 우선순위가 낮을 경우
                *postexpr++ = pop();    //스택에 저장되어있는 연산자가 우선순위가 높으니 pop 해서 postexpr에 대입
                *postexpr++ = ' ';  //각 연산자 사이에 공백을 추가
            }
            push(*expr);    //위 문장이 끝나면 expr의 연산자를 스택에 추가한다.
            expr++; //모든 작업이 끝나고 expr 1 증가
        }

        else expr++; //실수, 괄호, 연산자에 포함이 되지 않을 경우 무시하고 expr 위치 1 증가시켜 다음 문자를 읽는다.
    }

    while (!is_empty()) {   //마지막에는 스택에 남아있는 연산자들을 pop하여 postexpr에 추가하면 끝
        *postexpr++ = pop();
        *postexpr++ = ' ';  //각 연산자 사이에 공백 추가
    }

    postexpr--;
    *postexpr == '\0';  //마지막 연산자까지 모두 출력하게 되면 수식의 끝에 공백이 들어가게 되는데, 이를 NULL로 바꿔준다
}
```

결과는 다음과 같이 모든 실수들과 연산자 사이에 공백이 추가된 모습을 볼 수 있다.

```
34 23.4 * 37 83.2 46 - / +
72.5 34.5 / 23 32 78 - * +
35.3 45.6 42.6 35.7 / * - 47.4 / 25.6 74 46 + / +
56.2 33.3 * 23 124.5 46 - * +
89.5 23.4 * 74 32 46 + * +
```

(3)`double calc_postfix(char expr[])`

다음은 후위 표기식을 계산하는 함수인데, 앞서 후위 표기식으로 변환할 때 공백을 이용하여 구분했기 때문에, 여기서도 공백을 이용하여 실수와 실수, 실수와 연산자를 구분하는 알고리즘을 생각하였다. 또한 문자열로부터 입력받은 것을 실수로 변환하는 과정이 필요한데, 이는 buffer 배열 선언 후 atof()함수를 사용하여 변환하고 memset()함수로 초기화하는 과정으로 해결하였다.

(1) 문자열을 한 문자씩 읽는다.

(2) 읽은 문자가 숫자이거나 점일 경우, buffer에 우선 집어넣는다.

(3) 만약 c가 공백이고, 그 이전의 문자가 실수였을 경우는 실수 입력이 끝나고 공백이 추가된 결과임을 알 수 있다. 이렇게 조건을 하는 이유는, 연산자 사이에도 공백이 있어서 그것과 구분하기 위해서이다! 예를 들어 12.4공백+12.5라는 수식이 있으면, 12.4 다음 공백을 읽었을 때 공백 이전의 문자가 숫자이기 때문에 12.4를 하나의 실수로 받아들이는 것이다. 위 조건을 통해 실수의 입력이 끝났음을 알면, buffer에 저장된 배열을 실수로 바꿔주는 atof()함수를 사용하여 스택에 삽입한다.

(4) 실수로 변환하는 과정에서 사용한 buffer를 memset()함수를 사용해 초기화해야 다음 실수를 읽을 때 오류가 생기지 않는다.

(5) 만약 읽어들인 문자가 연산자일 경우, 스택에서 두번 pop()하여 계산하고, 계산 결과를 다시 스택에 저장한다.

(6) 위 조건들을 만족하지 않을 경우(예를 들어 연산자 사이에 있는 공백 등) 반복문을 계속한다.

(7) 수식을 끝까지 읽은 후 스택에 남아있는 값이 최종 계산 결과인데, 이를 pop()한다.

위 과정을 거쳐 완성된 함수는 다음과 같다.

```c
/*후위 표기 수식 계산 함수*/
double calc_postfix(char expr[]) {
    double val, val1, val2;
    int i = 0;  //expr을 스캔하기 위한 변수
    int j = 0;  //buffer에 문자를 입력하기 위한 변수
    char buffer[100] = { '0', };    //실수를 입력받을 buffer, 모든 문자를 NULL이 아닌 '0'으로 초기화
    char c; //expr을 한 문자씩 스캔할 때 사용하는 변수

    init_stack();
    while (expr[i] != '\0') {   //expr의 끝을 만나기 전 까지 반복
        c = expr[i++];  //우선 한 글자씩 c에 대입한다

        if ((c >= '0' && c <= '9') || c == '.') { buffer[j++] = c; }    //만약 c가 0과 9 사이의 숫자이거나 점일 경우 우선 버퍼에 대입한다
        else if (c == ' ' && (expr[i - 2] >= '0' && expr[i - 2] <= '9')) {  //만약 c가 공백이며 그 이전의 문자가 숫자로 끝났을 경우는? 한 실수의 스캔이 끝났다는것을 의미한다
            val = atof(buffer); //버퍼의 내용을 atof()함수를 이용해 double형 변수로 val에 대입
            push(val); //val을 스택에 삽입
            j = 0;
            memset(buffer, 0, strlen(buffer) * sizeof(char));   //버퍼를 새로 사용하기 위해 j를 0으로 바꾸고 memset()함수로 버퍼의 모든 내용을 0으로 초기화
        }
        else if (c == '+' || c == '-' || c == '*' || c == '/') {    //연산자를 만났을 경우 pop 2회 실행해서 계산한 값을 스택에 저장
            val2 = pop();   //먼저 스택에서 나온 값이 뒤로 가야한다
            val1 = pop();
            switch (c) {
            case '+':push(val1 + val2); break;
            case '-':push(val1 - val2); break;
            case '*':push(val1 * val2); break;
            case '/':push(val1 / val2); break;
            }
        }
        else continue;  //위 조건을 만족하지 않으면 반복문을 계속하여 다음 문자를 읽음
    }
    return pop();   //모든 반복이 끝나면 스택에는 최종 계산 결과가 남아있는데, 이를 pop해서 계산 결과를 반환
}
```

스택을 이용한 전자계산기 Result

Main()함수의 argument에 실행파일의 경로, txt파일의 경로, check 번호를 입력하여 스택으로 계산기가 동작한다. 스택을 사용했기 때문에, equation.txt에 있는 수식의 결과가 반대로 출력된다. 이 이유는 스택은 후입선출의 구조를 갖고 있기 때문이다!

```
C:₩Users₩user₩source₩repos₩datastructure₩x64₩Debug>datastructure.exe C:₩Users₩user₩Desktop₩equation.txt 1
calculator using stack
Result = 796.594624
Result = -1055.898551
Result = -0.189904
Result = 3676.960000
Result = 7866.300000
```

3. 큐를 이용한 전자계산기 구현

큐를 이용한 전자계산기를 구현하기 앞서, 필요한 것들을 생각했다.

1) 수식을 입력받아 괄호 검사 후 순서대로 enqueue()
2) 큐에서 front부터 순서대로 수식을 dequeu()한다
3) 후위 표기식으로 변환하거나 후위 표기식을 계산하는 과정은 앞서 서술한 함수를 사용하면 된다

앞의 스택을 이용한 전자계산기와 다른 점은, 수식을 입력받을 때 큐를 사용하여 선입선출의 방식으로 결과를 출력하는 것이지, 수식을 후위 표기식으로 변환하거나 후위 표기식을 계산하는 과정은 동일하다 따라서 큐를 이용한 전자계산기를 구현하기 위해서는, 소스에 연결리스트를 이용한 큐를 구현하고, main()에 check값이 0일 때 큐를 사용하여 수식을 입력받는 과정을 추가하면 완성이다.

(1) 연결리스트를 이용한 큐의 구현

```
//연결리스트를 이용한 큐의 구현
typedef char* Element_queue;
typedef struct LinkedNode_queue {
	Element_queue data;
	struct LinkedNode_queue* link;
} Node_queue;
//front와 rear의 선언
Node_queue* front = NULL, * rear = NULL;
//큐가 비어있는지 확인하는 함수
int is_empty_queue() { return front == NULL; }
//큐를 초기화하는 함수
void init_queue() { front = rear = NULL; }
//큐 안의 요소의 개수를 반환하는 함수
int size_queue() {
	Node_queue* p;
	int count = 0;
	for (p = front; p != NULL; p = p->link) count++;
	return count;
}
//큐 삽입 연산
void enqueue(Element_queue e) {
	Node_queue* p = (Node_queue*)malloc(sizeof(Node_queue));
	p->data = e;
	p->link = NULL;

	if (is_empty_queue()) front = rear = p;
	else {
		rear->link = p;
		rear = p;
	}
}
//큐 삭제 연산
Element_queue dequeue() {
	Node_queue* p;
	Element_queue e;
```

```
        if (is_empty_queue())
                error("큐 공백 오류");
        p = front;
        front = front->link;
        if (front == NULL) rear = NULL;
        e = p->data;
        free(p);
        return e;
}
//큐 peek 연산
Element_queue peek_queue() {
        if (is_empty_queue())
                error("큐 공백 오류");
        return front->data;
}
```

큐를 이용한 전자계산기 Result

```
C:₩Users₩user₩source₩repos₩datastructure₩x64₩Debug>datastructure.exe C:₩Users₩user₩Desktop₩equation.txt 0
calculator using queue
Result = 7866.300000
Result = 3676.960000
Result = -0.189904
Result = -1055.898551
Result = 796.594624
```

앞의 스택을 사용했을 때랑 다른 점은, **스택은 후입선출 방식이고 큐는 선입선출** 방식이기에 결과가 나오는 순서가 서로 반대이다.

4. 최종 결과(첨부된 ds_20190895.c 와 동일한 소스)

```
C:₩Users₩user₩source₩repos₩datastructure₩x64₩Debug>datastructure.exe C:₩Users₩user₩Desktop₩equation.txt 0
calculator using queue
Result = 7866.300000
Result = 3676.960000
Result = -0.189904
Result = -1055.898551
Result = 796.594624

C:₩Users₩user₩source₩repos₩datastructure₩x64₩Debug>datastructure.exe C:₩Users₩user₩Desktop₩equation.txt 1
calculator using stack
Result = 796.594624
Result = -1055.898551
Result = -0.189904
Result = 3676.960000
Result = 7866.300000
```

*보고서에 소스코드를 작성하긴 했지만, 가독성이 좋지 못하여 함께 첨부한 소스파일을 실행시켜 확인하시길 바랍니다.

```c
#define _CRT_SECURE_NO_WARNINGS    //Visual C에서 fopen함수 사용 시 보안 오류를 출력하지 않기 위해 사용. Linux, OS X에서는 선언하지 않아도 된다.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>



//에러 출력 함수
void error(char* str)
{
	fprintf(stderr, "%s\n", str);
	exit(1);
}



//연결리스트 이용한 스택
typedef double Element;
typedef struct LinkedNode {
	Element data;
	struct LinkedNode* link;
} Node;
Node* top = NULL;
//스택이 비어있는지 확인하는 함수
int is_empty() { return top == NULL; }
//스택 초기화 함수
void init_stack() { top = NULL; }
//스택 안의 요소의 개수를 반환하는 함수
int size(){
	Node* p;
	int count = 0;
	for (p = top; p != NULL; p = p->link) count++;
	return count;
}
//스택 삽입 연산
void push(Element e){
	Node* p = (Node*)malloc(sizeof(Node));
```

```
        p->data = e;
        p->link = top;
        top = p;
}
//스택 삭제연산
Element pop(){
        Node* p;
        Element e;
        if (is_empty())
                error("스택 공백 에러");
        p = top;
        top = p->link;
        e = p->data;
        free(p);
        return e;
}
//스택 peek연산
Element peek(){
        if (is_empty())
                error("스택 공백 에러");
        return top->data;
}

/*문자열을 저장하기 위해 선언한 연결리스트를 이용한 스택2
기존의 스택 연산과 동일하며 노드 안에 문자열을 저장할 수 있게 하였다*/
typedef struct LinkedNode2 {
        char* str;
        struct LinkedNode2* link2;
}Node2;
Node2* top2 = NULL;
int is_emptystr() { return top2 == NULL; }
void init_stackstr() { top2 = NULL; }
void pushstr(char* expr) {
        Node2* p = (Node2*)malloc(sizeof(Node2));
        p->str = expr;
        p->link2 = top2;
        top2 = p;
}
char* popstr() {
        Node2* p;
        char* str;
        if (is_emptystr())
                error("스택 공백 에러");
        p = top2;
        top2 = p->link2;
        str = p->str;
        free(p);
        return str;
}


//연결리스트를 이용한 큐의 구현
typedef char* Element_queue;
typedef struct LinkedNode_queue {
        Element_queue data;
        struct LinkedNode_queue* link;
```

```
} Node_queue;
//front와 rear의 선언
Node_queue* front = NULL, * rear = NULL;
//큐가 비어있는지 확인하는 함수
int is_empty_queue() { return front == NULL; }
//큐를 초기화하는 함수
void init_queue() { front = rear = NULL; }
//큐 안의 요소의 개수를 반환하는 함수
int size_queue() {
	Node_queue* p;
	int count = 0;
	for (p = front; p != NULL; p = p->link) count++;
	return count;
}
//큐 삽입 연산
void enqueue(Element_queue e) {
	Node_queue* p = (Node_queue*)malloc(sizeof(Node_queue));
	p->data = e;
	p->link = NULL;

	if (is_empty_queue()) front = rear = p;
	else {
		rear->link = p;
		rear = p;
	}
}
//큐 삭제 연산
Element_queue dequeue() {
	Node_queue* p;
	Element_queue e;
	if (is_empty_queue())
		error("큐 공백 오류");
	p = front;
	front = front->link;
	if (front == NULL) rear = NULL;
	e = p->data;
	free(p);
	return e;
}
//큐 peek 연산
Element_queue peek_queue() {
	if (is_empty_queue())
		error("큐 공백 오류");
	return front->data;
}




/*괄호 검사 함수
	조건 1 : 왼쪽 괄호의 개수와 오른쪽 괄호의 개수는 같아야 함
	조건 2 : 같은 괄호는 왼쪽에서 먼저 나와야 함
	조건 3 : 서로 다른 타입의 괄호가 짝을 이루면 안됨
괄호에 오류가 없는 경우 0을 반환
괄호에 오류가 있는 경우 1, 2, 3 반환*/
```

```
int check_matching(char expr[]) {
	int i = 0, prev;
	char ch; //문자열을 하나씩 읽을 때 문자를 저장하는 변수

	init_stack();
	while (expr[i] != '\0') {
		ch = expr[i++];  //문자열을 index i 부터 읽어들이고, i+1을 실행
		if (ch == '[' || ch == '{' || ch == '(')
			push(ch);	//읽어들인 문자가 왼쪽 괄호일 경우 스택에 읽어들인 왼쪽 괄호를 push
		else if (ch == ']' || ch == '}' || ch == ')') {
			if (is_empty())
				return 2;	//조건 2 위반 : 오른쪽 괄호가 왼쪽 괄호보다 먼저 나옴
			prev = pop();	//스택이 비어있지 않은 상태에서 top에 있는 괄호를 꺼내서 prev에 저장
			if ((ch == ']' && prev != '[') || (ch == '}' && prev != '{') || (ch == ')' && prev !=
'(')) {
				return 3;			//문자열을 읽다가 ch에 저장된 오른쪽 괄호와,
스택에서 pop한 prev의 왼쪽 괄호가 같지 않을 경우 조건 3 위반 : 괄호의 쌍이 맞지 않음
			}
		}
	}
	if (is_empty() == 0) return 1;	//조건 1 위반 : 스택이 비어있지 않는 경우 좌괄호와 우괄호의 개수가 같지
않다
	return 0;	//위 조건에 걸리지 않으면 수식의 괄호에 오류가 없음을 알리며 0 반환
}

//연산자의 우선순위를 구하는 함수. 우선순위가 높을 수록 숫자가 크다.
int precedence(char op) {
	switch (op) {
	case '(':case')':return 0;
	case '+':case'-':return 1;
	case '*':case'/':return 2;
	}
	return -1;
}

/*중위 표기 수식을 후위 표기 수식으로 변환하는 함수
한 자리의 숫자가 아니기 때문에 실수와 실수, 실수와 연산자를 구분할 수 있도록 각각의 사이에 공백을 추가해 구분한다.*/
void infix_to_postfix(char* expr, char* postexpr) {
	init_stack();
	while (*expr != '\0') {	//문자열의 끝을 만나기 전 까지 반복

		if (*expr >= '0' && *expr <= '9') {//조건문 : 실수를 읽기 위한 조건문. 숫자나 점을 만났을 경우
			do {
				*postexpr++ = *expr++;	//postexpr에 expr의 값을 대입하고 각각 포인터가
가리키는 위치 1 증가하면 다음 글자를 계속 읽을 수 있다
			} while ((*expr >= '0' && *expr <= '9') || *expr == '.');	//do-while문을 이용해 계속
postexpr에 대입한다
			*postexpr++ = ' ';	//실수를 다 읽으면 postexpr에 공백을 추가
		}

		else if (*expr == '(') {	//조건문 : 좌괄호를 만났을 경우
			push(*expr++);	//우선 스택에 저장하고 expr의 위치 증가
		}

		else if (*expr == ')') {	//조건문 : 우괄호를 만났을 경우
			while (peek() != '(') {	//좌괄호를 만나기 전 까지(반복문 안에서 pop을 하기 때문에
```

```
peek을 사용해도 무한 반복되지 않는다)
                                *postexpr++ = pop();      //스택에 저장된 연산자를 pop 해서 postexpr에 대입한다
                                *postexpr++ = ' ';        //연산자를 pop 하면서 연산자 사이에 공백을 추가
                        }
                        pop();  //좌괄호를 스택에서 pop한다
                        expr++;  //괄호 안의 연산자를 다 꺼냈을 경우 expr 위치 증가
                }

                else if (*expr == '+' || *expr == '-' || *expr == '*' || *expr == '/') {        //조건문 :
연산자를 만났을 경우
                        while (!is_empty() && (precedence(*expr) <= precedence(peek()))) {     //스택이 비어있지
않고, expr의 연산자가 스택에 저장된 연산자보다 우선순위가 낮을 경우
                                *postexpr++ = pop();      //스택에 저장되어있는 연산자가 우선순위가 높으니 pop
해서 postexpr에 대입
                                *postexpr++ = ' ';        //각 연산자 사이에 공백을 추가
                        }
                        push(*expr);      //위 문장이 끝나면 expr의 연산자를 스택에 추가한다.
                        expr++;  //모든 작업이 끝나고 expr 1 증가
                }

                else expr++; //실수, 괄호, 연산자에 포함이 되지 않을 경우 무시하고 expr 위치 1 증가시켜 다음
문자를 읽는다.
        }

        while (!is_empty()) {     //마지막에는 스택에 남아있는 연산자들을 pop하여 postexpr에 추가하면 끝
                *postexpr++ = pop();
                *postexpr++ = ' ';        //각 연산자 사이에 공백 추가
        }

        postexpr--;
        *postexpr == '\0';        //마지막 연산자까지 모두 출력하게 되면 수식의 끝에 공백이 들어가게 되는데, 이를
NULL로 바꿔준다
}




/*후위 표기 수식 계산 함수*/
double calc_postfix(char expr[]) {
        double val, val1, val2;
        int i = 0;        //expr을 스캔하기 위한 변수
        int j = 0;        //buffer에 문자를 입력하기 위한 변수
        char buffer[100] = { '0', };        //실수를 입력받을 buffer, 모든 문자를 NULL이 아닌 '0'으로 초기화
        char c; //expr을 한 문자씩 스캔할 때 사용하는 변수

        init_stack();
        while (expr[i] != '\0') {  //expr의 끝을 만나기 전 까지 반복
                c = expr[i++];    //우선 한 글자씩 c에 대입한다

                if ((c >= '0' && c <= '9') || c == '.') { buffer[j++] = c; }  //만약 c가 0과 9 사이의 숫자이거나
점일 경우 우선 버퍼에 대입한다
                else if (c == ' ' && (expr[i - 2] >= '0' && expr[i - 2] <= '9')) {     //만약 c가 공백이며 그
이전의 문자가 숫자로 끝났을 경우는? 한 실수의 스캔이 끝났다는것을 의미한다
                        val = atof(buffer);       //버퍼의 내용을 atof()함수를 이용해 double형 변수로 val에 대입
                        push(val); //val을 스택에 삽입
```

```
                        j = 0;
                        memset(buffer, 0, strlen(buffer) * sizeof(char));    //버퍼를 새로 사용하기 위해 j를 0으로 바꾸고 memset()함수로 버퍼의 모든 내용을 0으로 초기화
                }
                else if (c == '+' || c == '-' || c == '*' || c == '/') {      //연산자를 만났을 경우 pop 2회 실행해서 계산한 값을 스택에 저장
                        val2 = pop();     //먼저 스택에서 나온 값이 뒤로 가야한다
                        val1 = pop();
                        switch (c) {
                        case '+':push(val1 + val2); break;
                        case '-':push(val1 - val2); break;
                        case '*':push(val1 * val2); break;
                        case '/':push(val1 / val2); break;
                        }
                }
                else continue;    //위 조건을 만족하지 않으면 반복문을 계속하여 다음 문자를 읽음
        }
        return pop();     //모든 반복이 끝나면 스택에는 최종 계산 결과가 남아있는데, 이를 pop해서 계산 결과를 반환
}



//main함수 실행. argument를 입력받아 argv[2]=0이면 큐를 사용해 수식을 저장, argv[2]=1이면 스택을 사용해 수식을 저장
int main(int argc, char* argv[]) {
        FILE* fp = NULL;  //파일 포인터 선언
        int check;        //argv[2] 를 저장하기 위한 변수
        int i;
        char** equation_f = NULL, ** input_equation = NULL; //입력받을 수식을 저장할 두 개의 char이중포인터 선언
        int count = 0;    //문자열이 몇 줄인지 세기 위한 함수

        if (argc != 3)    error("exec equation check(0 or 1)");       //argument count가 3이 아닐 경우는 명령어가 잘못 입력된 경우이다

        fp = fopen(argv[1], "rt"); //argument value의 두 번째로 입력받은 파일 경로를 텍스트 읽기 모드로 오픈
        check = atoi(argv[2]);     //argument value의 세 번째로 입력받은 내용을 atoi()함수를 사용해 정수로 변환 후 check에 대입

        //아래는 이중 포인터를 동적할당하는 과정이다. 배열의 내용을 초기화하기 위해 calloc()함수를 사용하였다
        equation_f = (char**)calloc(sizeof(char*) * 10, sizeof(char*));
        for (i = 0; i < 10; i++)
                equation_f[i] = (char*)calloc(sizeof(char) * 256, sizeof(char));

        input_equation = (char**)calloc(sizeof(char*) * 10, sizeof(char*));
        for (i = 0; i < 10; i++)
                input_equation[i] = (char*)calloc(sizeof(char) * 256, sizeof(char));

        //파일 경로가 잘못되었을 경우
        if (fp == NULL)
                error("fopen is failed");

        if (check == 0) { //0번 옵션일 경우 큐를 이용한 계산기 실행
                printf("calculator using queue\n");

                for (i = 0; fgets(input_equation[i], 256, fp) != NULL; i++) { //txt파일로부터 끝이 아닐 때 까지 문자열을 한 줄씩 입력받음
                        if (check_matching(input_equation[i]) != 0) //가장 먼저 괄호검사를 실행한다
```

```
                                error("괄호 검사 실패");
                        enqueue(input_equation[i]);         //순서대로 문자열을 큐에 저장하고, 몇 줄을
입력받았는지 count
                        count++;
                }
                for (i = 0; i < count; i++) {
                        infix_to_postfix(dequeue(), equation_f[i]); //dequeue_str()함수로 한 줄씩 꺼내 계산 후
결과를 출력한다.
                        printf("Result = %lf\n", calc_postfix(equation_f[i]));
                }
        }

        else if (check == 1) {     //1번 옵션일 경우 스택을 이용한 계산기 실행
                printf("calculator using stack\n");

                for (i = 0; fgets(input_equation[i], 256, fp) != NULL; i++) { //txt파일로부터 끝이 아닐 때 까지
문자열을 한 줄씩 입력받음
                        if (check_matching(input_equation[i]) != 0) //가장 먼저 괄호검사를 실행한다
                                error("괄호 검사 실패");
                        pushstr(input_equation[i]);         //순서대로 문자열을 스택에 저장하고, 몇 줄을
입력받았는지 count
                        count++;
                }
                for (i = 0; i < count; i++) {
                        infix_to_postfix(popstr(), equation_f[i]);  //popstr()함수로 한 줄씩 꺼내 계산 후 결과를
출력한다.
                        printf("Result = %lf\n", calc_postfix(equation_f[i]));
                }
        }

        //할당된 메모리를 모두 해제하고 열었던 파일을 닫는다
        for (i = 0; i < 10; i++)
                free(equation_f[i]);
        free(equation_f);
        for (i = 0; i < 10; i++)
                free(input_equation[i]);
        free(input_equation);
        fclose(fp);

        return 0;
}
```