KWANGWOON UNIVERSITY

# 자 료 구 조

## - Homework #2 -

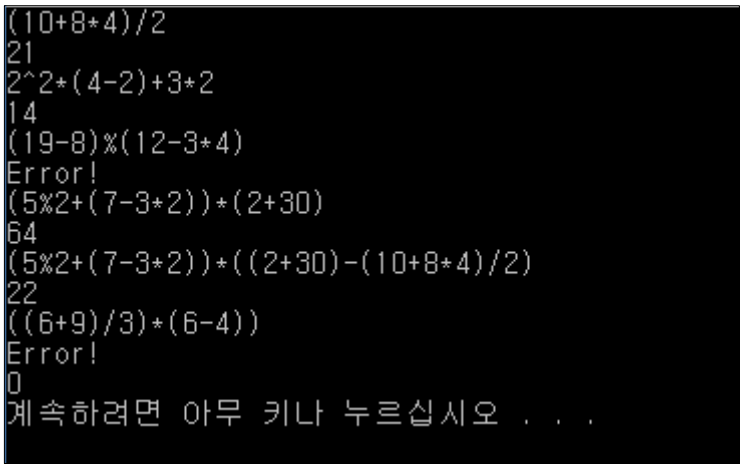
( Stack Calculator )

* Visual Studio 2015 environment




담당 교수님 : 김용혁 교수님
제 출 날 짜 : 2017. 11. 07.
학 과 : 컴퓨터소프트웨어학과
학 번 : 2014707040
이 름 : 유 진 혁

# 1. 프로그램에 대해




<그림 1. 프로그램 실행 창>

이 프로그램은 C++ STL의 스택을 활용하여 일반적인 정수 수식의 결과를 line-by-line 방식으로 출력하는 프로그램이다. 프로그램에 0을 입력하면 프로그램이 종료가 되고, 잘못된 괄호나 0으로 나누는 연산이 포함된 수식을 입력하면 "Error!"가 출력된다. 이외의 오류에 대해선 예외처리하지 않는다. 오류 발생 여부 판단을 위해 err_occurred라는 bool형 전역변수가 사용된다.

Numeric이라는 사용자 정의 구조체가 사용됐고, 이를 이용하는 transPostfix와 evalPostfix라는 사용자 정의 함수가 사용되었다.

# 2. Numeric 구조체

```
typedef struct { // 수 또는 문자 표현 구조체
    char symbol; // 숫자, 연산자, 괄호 등의 문자
    int digits = 0; // 숫자의 자릿수 (0 == 연산자나 괄호, 1 == 일의 자릿수, 2 == 십의 자릿수, ...)
} Numeric;
```

<그림 2. Numeric 구조체>

프로그램에 숫자가 입력 될 때 해당 숫자에 자릿수 정보까지 담기 위해 Numeric이라는 구조체를 만들었다. 멤버 변수로는 입력 문자를 담는 char형 변수 symbol, 자릿수 정보를 담는 int형 변수 digits가 있다. 변수 digits에는 0이 기본 값으로 저장된다. symbol 변수에 저장된 문자가 숫자라면 transPostfix 함수 내에서 digits의 값을 자릿수에 맞게 증가시킨다. digits의 값이 1이라면 symbol의 숫자는 일의 자리의 숫자인 것이고, 2이라면 십의 자리의 숫자라는 뜻이다. 이후 자릿수에 대해서도 마찬가지이다. symbol에 담긴 문자가 연산자나 괄호 문자라면 digits의 값은 그대로 0이 된다.

# 3. transPostfix 함수

```
#include <stack>
#include <cstring>
using namespace std;

Numeric* transPostfix(char infix[], size_t& postfix_len) // infix notation을 postfix notation으로 변환하는 함수
```

<그림 3. transPostfix 함수 Prototype>

transPostfix 함수는 Infix notation을 Postfix notation으로 변환해주는 함수이다.

transPostfix 함수를 사용하기 위해선 <stack>과 <cstring>을 include 해주어야 한다. 함수 내부에서 스택이 만들어지고 strlen 함수가 사용되기 때문이다.

transPostfix 함수는 인자로 char형 배열(포인터) infix를 받고, size_t형 변수 postfix_len을 reference로 받는다. infix는 Postfix notation으로 바꾸고 싶은 식이 담긴 문자열이다. 매개변수로 받은 postfix_len의 값을 바꾸기 위해 call-by-reference 방법을 사용했다. 함수 호출 후 postfix_len에는 postfix 식의 길이 값이 저장된다.

함수의 반환 값은 Numeric 구조체의 포인터이다. 정확히는, 동적 할당된 Numeric 구조체 타입의 배열을 가리키는 포인터이다. Postfix notation 식으로 계산을 하기 위해선 Postfix 식 문자열의 숫자들이 몇 번째 자릿수인지 알아야 하기 때문에 자릿수 정보를 같이 담을 수 있는 Numeric 구조체 배열을 반환한다.

함수의 세부 내용은 아래와 같다.

```
stack<char> stck;
Numeric* postfix = new Numeric[1000]; // postfix 식을 담을 배열 동적 할당
postfix_len = 0; // postfix 식 문자의 길이 (인자로 받은 변수)
```

<그림 4. transPostfix 함수의 변수 초기화 부분>

먼저 스택을 만들고, postfix 식이 들어갈 Numeric 구조체 배열을 main 함수에서도 계속 유지시키기 위해 동적 할당을 했다. 그리고 인자로 받은 postfix_len을 우선 0으로 초기화해주었다. infix 배열의 요소를 처음부터 끝까지 하나하나 확인하기 위해 반복문을 돌렸다. 반복문 안에는 다음과 같은 조건 분기를 만들었다.

```
if (infix[i] == '(') // 왼쪽 괄호일 경우
    stck.push(infix[i]);
```

<그림 5. infix 식의 문자가 '('일 경우>

먼저, infix 배열의 i번째 요소가 '(' (왼쪽 괄호)일 경우 스택에 push 해준다.

# 3. transPostfix 함수

```
else if (infix[i] >= '0' && infix[i] <= '9') // 숫자일 경우
{
    postfix[postfix_len].symbol = infix[i]; // postfix 배열에 넣음

    // infix 식의 현재 문자와 이전 문자들을 확인하여 자릿수를 알맞게 증가
    for (size_t iter = 0; (iter <= postfix_len) && (infix[i - iter] >= '0') && (infix[i - iter] <= '9'); iter++)
        postfix[postfix_len - iter].digits++;

    postfix_len++; // postfix 식 문자 길이 증가
}
```

<그림 6. infix 식의 문자가 숫자일 경우>

infix 배열의 i번째 요소가 '0'부터 '9'까지의 숫자일 경우 해당 문자가 postfix 마지막 요소의 문자 변수 symbol에 저장된다. 그리고 반복문을 통해 infix 배열의 이전 요소를 확인하여 자릿수 정보를 알맞게 postfix 배열에 저장하도록 하였다. 우선 postfix 배열에 저장된 현재 요소의 digits 값을 1 증가시켜주고, infix 배열의 이전 요소를 확인하여 그것이 숫자라면 이전 요소의 digits 값도 1 증가시켜줬다. 이 과정을 반복하여 일의 자릿수 숫자엔 digits가 1, 십의 자릿수 숫자엔 2, 백의 자릿수 숫자엔 3, 이런 식으로 자릿수에 따라 1씩 증가되도록 하였다. 자릿수 정보 변경이 끝나면, postfix 배열의 길이가 1 증가하였으므로 postfix_len 변수도 1 증가된다.

```
else if (infix[i] == '^') // ^ 연산자일 경우 (가장 높은 우선순위)
{
    // 스택이 비거나, 스택의 top이 왼쪽 괄호나 낮은 우선순위의 연산자일 때까지 반복
    while (!stck.empty() && stck.top() != '(' && stck.top() != '*' && stck.top() != '/' && stck.top() != '%' && stck.top() != '+' && stck.top() != '-')
    {
        postfix[postfix_len++].symbol = stck.top(); // 스택의 top을 postfix 배열에 넣음
        stck.pop();
    }
    stck.push(infix[i]);
}

else if (infix[i] == '*' || infix[i] == '/' || infix[i] == '%') // * / % 연산자일 경우
{
    while (!stck.empty() && stck.top() != '(' && stck.top() != '+' && stck.top() != '-')
    {
        postfix[postfix_len++].symbol = stck.top();
        stck.pop();
    }
    stck.push(infix[i]);
}

else if (infix[i] == '+' || infix[i] == '-') // + - 연산자일 경우 (가장 낮은 우선순위)
{
    while (!stck.empty() && stck.top() != '(')
    {
        postfix[postfix_len++].symbol = stck.top();
        stck.pop();
    }
    stck.push(infix[i]);
}
```

<그림 7. infix 식의 문자가 연산자일 경우>

그 다음, infix 배열의 i번째 요소가 연산자일 경우이다. 연산자 중에서도 우선순위가 같은 것끼리 조건 분기를 나누었다. 스택이 비어있지 않고 스택의 top이 왼쪽 괄호나 낮은 우선순위의 연산자가 아니라면, 스택의 top을 postfix 배열에 넣고 postfix_len 값을 증가시킨 뒤, 스택 pop 해주기를 반복하였다. 반복문을 진행하지 않거나 더 이상 반복 조건이 맞지 않아 반복문을 나오게 되면 infix의 i번째 요소가 스택에 push 된다.

# 3. transPostfix 함수

```
else if (infix[i] == ')') // 오른쪽 괄호일 경우
{
    while (!stck.empty() && stck.top() != '(') // 스택이 비거나 스택의 top이 왼쪽 괄호일 때까지 반복
    {
        postfix[postfix_len++].symbol = stck.top(); // 스택의 top을 postfix 배열에 넣음
        stck.pop();
    }

    if (stck.empty()) // 스택이 비었다면 (오른쪽 괄호가 왼쪽 괄호보다 많을 경우)
    {
        err_occurred = true; // 에러 발생
        return postfix; // 함수 종료
    }

    stck.pop(); // 스택 top에 있는 왼쪽 괄호 pop
}
```

<그림 8. infix 식의 문자가 ')'일 경우>

마지막으로 infix 배열의 i번째 요소가 ')' (오른쪽 괄호)일 경우, 스택이 비거나 스택의 top이 왼쪽 괄호일 때까지 스택의 top을 postfix 배열에 넣고 postfix_len을 증가시킨 뒤, pop 해주기를 반복하였다. 이 과정이 끝나고 스택에 왼쪽 괄호가 남지 않고 비어있다면 infix 식에 오른쪽 괄호가 왼쪽 괄호보다 많다는 뜻이므로, err_occurred 변수를 true로 바꾸고 함수가 종료된다. 정상적으로 스택에 왼쪽 괄호가 남아있다면 남아있는 왼쪽 괄호가 pop 된다.

```
while (!stck.empty()) // 스택에 문자가 남아있는 경우
{
    if (stck.top() == '(') // 스택 top이 왼쪽 괄호라면 (왼쪽 괄호가 오른쪽 괄호보다 많은 경우)
    {
        err_occurred = true; // 에러 발생
        return postfix;
    }

    postfix[postfix_len++].symbol = stck.top(); // 남은 문자 postfix 배열에 넣음
    stck.pop();
}

return postfix;
```

<그림 9. infix의 문자 전부 확인 후 스택의 문자가 남아있는 경우>

infix 배열의 마지막 요소까지 확인한 뒤, 스택에 문자가 남아있다면 남아있는 문자를 모두 postfix 배열에 넣고 pop 해주었다. 이에 따라 postfix_len도 같이 증가시켜주었다. 이 때, 스택에 왼쪽 괄호가 남아있다면, infix 식에서 왼쪽 괄호가 오른쪽 괄호보다 많다는 뜻이므로 오류 표시를 해주고 함수가 종료된다.

이렇게 완성된 postfix 배열이 반환되고 함수가 종료된다.

# 4. evalPostfix 함수

```
#include <stack>
#include <cmath>
using namespace std;

int evalPostfix(Numeric postfix[], size_t len) // postfix 식을 계산하는 함수
```

<그림 10. evalPostfix 함수 Prototype>

evalPostfix 함수는 Postfix notation 식을 계산하여 결과 정수 값을 반환하는 함수이다.

evalPostfix 함수를 사용하기 위해선 <stack>과 <cmath>을 include 해주어야 한다. 함수 내부에서 스택이 만들어지고 pow 함수가 사용되기 때문이다.

evalPostfix 함수는 인자로 Numeric 구조체 배열 postfix와 size_t형 변수 len을 받는다. postfix는 Postfix notation 식의 문자와 자릿수 정보가 차례로 저장된 Numeric 구조체 배열이다. len은 postfix 배열의 길이 값이다.

정수 연산만을 하기 때문에 결과 값도 정수이고 int형을 반환한다.

함수의 세부 내용은 아래와 같다.

```
stack<int> stck;
int num = 0; // 문자에서 변환된 수 저장 변수
int result; // 연산 결과 값 저장 변수
```

<그림 11. transPostfix 함수의 변수 초기화 부분>

먼저 스택을 만든다. Numeric 구조체는 숫자를 문자로 저장하고 있으므로 이를 정수로 바꾸는 과정이 필요하다. 또, 자릿수 정보를 참고하여 자릿수 별로 분리된 숫자들을 모아 하나의 정수로 만드는 과정도 필요하다. 이러한 과정을 거쳐 만들어진 정수가 저장될 변수 num을 선언하고 0으로 초기화해줬다. 연산의 결과 값이 저장될 변수 result도 선언해주었다. 이제 반복문을 통해 postfix 배열의 symbol들을 하나씩 확인하는 과정에 들어간다.

```
if (postfix[i].symbol >= '0' && postfix[i].symbol <= '9') // 숫자일 경우
{
    if (postfix[i].digits == 1) // 일의 자리 수라면
    {
        num += postfix[i].symbol - '0'; // 문자를 해당 정수로 변환 후
        stck.push(num); // num에 저장된 수를 스택에 push
        num = 0; // 초기화
    }

    else // 십의 자리 이상의 수라면
        num += (postfix[i].symbol - '0') * (int)pow(10, postfix[i].digits - 1); // 자릿수에 맞게 정수 변환 후 값 저장
}
```

<그림 12. postfix 식의 문자가 숫자일 경우>

# 4. evalPostfix 함수

postfix 배열의 i번째 요소 symbol 문자가 숫자일 경우, 이를 자릿수까지 맞는 완전한 정수로 바꿔 주어야한다. 그래서 digits의 값이 1일 때와 아닐 때를 나누었다. 먼저 아닐 때, 십의 자리 이상의 수일 때를 보면, symbol 문자에 '0'을 빼주어 문자와 맞는 정수로 바꿔주었다. 거기에 10의 제곱수가 곱해져 자릿수에 맞게 값을 수정한 뒤 num 변수에 더하여 저장된다. 십의 자리 이상의 수들을 num에 다 저장하고 일의 자리의 수에 도달하면, 마찬가지로 문자와 맞는 정수로 바꾸고 num에 더해 대입해준 뒤 최종적으로 그 num가 스택에 push 된다. 그리고 num을 다시 0으로 초기화해줬다.

```
else // 연산자일 경우
{
    int num2 = stck.top();
    stck.pop();
    int num1 = stck.top();
    stck.pop();
```

<그림 13. postfix 식의 문자가 연산자일 경우>

postfix 배열의 i번째 요소 symbol 문자가 숫자가 아닐 경우, 즉, 연산자라면 top()과 pop()으로 스택에서 정수 두 개를 꺼내 각각 변수가 저장된다.

```
// 각 연산자에 맞게 연산
switch (postfix[i].symbol)
{
case '^':
    result = (int)pow(num1, num2);
    break;

case '*':
    result = num1 * num2;
    break;

case '/':
    if (num2 == 0) // 0으로 나누는 경우
    {
        err_occurred = true; // 에러 발생
        return 0; // 함수 종료
    }
    else
    {
        result = num1 / num2;
        break;
    }

case '%':
    if (num2 == 0)
    {
        err_occurred = true;
        return 0;
    }
    else
    {
        result = num1 % num2;
        break;
    }

case '+':
    result = num1 + num2;
    break;

case '-':
    result = num1 - num2;
    break;
}

stck.push(result);
```

<그림 14. 두 정수 연산>

그리고 해당 연산자에 맞게 두 정수를 계산한다. 그 결과는 스택에 push 한다. 이 때, 0으로 나누거나 0으로 나누어 나머지를 구하는 연산을 할 경우 예외처리를 하기 위해 / 연산과 % 연산 부분에 조건문을 두었다. "/0"이나 "%0"을 할 경우, err_occurred를 true로 바꾸고 함수가 바로 종료된다.

postfix 배열의 마지막 요소까지 확인하여 반복문을 마치면 스택의 top을 반환하며 함수가 종료된다. 이는 계산의 결과 값이고 결국 result에 저장된 값과 같은 값이다.

# 5. main 함수

```
int main()
{
    char infix[1000];
    Numeric* postfix;
    int result;
    size_t postfix_len;

    while (true)
    {
        cin >> infix; // 계산할 식 입력

        if (strcmp(infix, "0") == 0) // 0 입력 시 프로그램 종료
        {
            return 0;
        }

        postfix = transPostfix(infix, postfix_len);
        // transPostfix 함수 호출 후 postfix_len 변수에 postfix 식 길이 저장됨

        if (err_occurred) // transPostfix 함수 호출 후 에러 발생했다면
        {
            cout << "Error!" << endl;
            err_occurred = false;
        }
        else
        {
            result = evalPostfix(postfix, postfix_len);

            if (err_occurred) // evalPostfix 함수 호출 후 에러 발생했다면
            {
                cout << "Error!" << endl;
                err_occurred = false;
            }
            else
                cout << result << endl; // 계산 결과 출력
        }

        delete[] postfix; // transPostfix 함수에서 한 동적 할당 해제
    }
}
```

<그림 15. main 함수>

infix 식을 담을 문자열, postfix 식을 담을 구조체 배열, 연산 결과 값을 저장할 result 변수, postfix 배열의 길이 값을 저장할 postfix_len 변수를 선언했다.

그리고 무한루프를 만들어 infix에 0이 입력될 때까지 프로그램이 종료되지 않도록 했다. transPostfix 함수를 호출하고 반환 값이 postfix에 들어간다. 에러가 발생하지 않았을 때에만 evalPostfix 함수를 호출하여 계산 결과가 result에 저장된다. 마찬가지로 에러가 발생하지 않았을 경우에만 계산 결과를 출력하고 동적 할당을 해제한 뒤 다시 수식을 입력받는다. 에러가 발생했을 땐 결과 대신 “Error!”가 출력되고 err_occurred 변수가 다시 false로 돌아간다.

# 6. 고 찰

처음 과제에 대한 설명을 들었을 땐 간단한 과제라 생각했다. 강의 자료에 나와 있는 내용을 그대로 코드로 작성하면 끝이라 생각했기 때문이다. 하지만 강의 자료의 스택 계산기 방식과 과제에서의 스택 계산기 방식의 차이가 좀 있었다. 강의 자료에서는 결과를 바로 출력하는 반면 과제에선 postfix로 변환한 후 그 postfix 식을 이용하여 다시 계산을 해야 했다. 그렇기 때문에 postfix 식이 저장될 공간이 필요하다 생각했고 배열을 이용하게 되었다. 기본적으로 infix 식을 스택을 이용하여 계산하려면 postfix로 바꾸는 기능, postfix 식을 계산하는 기능이 필요하므로 이 두 가지 기능을 수행하는 함수를 각각 만들었다.

코드를 작성하면서 런타임 에러가 종종 발생하였는데 원인을 금방 찾지 못해 이를 디버깅하는데 많은 시간이 소요되었다. 처음에 postfix로 변환하는 함수 내부에서 배열을 생성하고 반환 값을 그 배열, 그러니까 postfix 문자열로 하려고 했더니, 런타임 에러가 발생하였다. 그냥 봤을 땐 함수의 전반적인 코드에는 문제가 없어서 한 줄씩 디버깅해야 원인을 찾을 수 있었다. 함수 내부에서 배열을 동적 할당함으로서 이 문제를 해결할 수 있었다.

infix 식을 postfix로 바꾸고 나서 계산을 하려고 보니 infix 일 때와 postfix 일 때 괄호가 사라져서 문자열의 길이가 달랐다. 때문에 한 문자씩 검사하는 반복문을 돌려야 되는데 길이를 알 수 없어 난감한 상황이 발생하였다. 이전 함수에서 postfix로 바꾸면서 길이도 함께 반환되면 좋겠다는 생각에 매개변수 하나를 레퍼런스로 받아 그 변수에 길이 값이 저장되도록 하였다.

이렇게 프로그램을 구성한 후 과제의 예시들을 입력해보았는데 프로그램에 큰 문제가 있었다. 한 자릿수 숫자만 연산이 가능한 것이다. 애초에 배열에 수들이 각 문자들로 저장되니 두 자리 이상의 수를 하나의 수로 인식하지 못하는 것이다. 그래서 각 숫자에 자릿수 정보가 같이 담기면 좋을 것 같다는 생각을 하였고 생각 끝에 구조체를 이용하기로 했다. 구조체 배열을 선언하고 각 숫자에 자릿수 정보가 알맞게 담기게끔, 이전 문자들도 참조하는 알고리즘을 짜다보니 꽤나 코드가 복잡해지고 생각하기도 어려웠다. 그래도 문자들을 정수 숫자로 표현하는데 이 방법이 가장 깔끔하고 최선이 아닐까 싶다.