스택과 계산기

HandyPost는 한 도영(HDNua)이 작성하는 포스트 문서입니다.

1. 개요

스택 자료구조가 무엇인지 이해하고, 이를 이용하여 복합적 사칙 연산이 가능한 계산기를 C++ 프로그래밍 언어로 만드는 것을 목표로 한다.

2. 프로젝트 준비

개발 환경은 C++ 프로그래밍 언어로 개발할 수 있는 어떤 것으로 골라도 좋다. 이 문서에서는 개발 도구의 사용 방법은 독자가 이미 익숙하여 스스로 할 수 있다고 가정한다. 혹 이 부분이 준비되지 않았다고 생각한다면 앞으로의 내용을 읽기 아주 곤란하기 때문에, 먼저 이에 관한 내용을 숙지한 상태여야 한다. 도구의 깊은 내용을 모두 알아야 하는 것이 아니고, 개발 도구를 이용하여 프로그램을 빌드 하는 방법만 정확히 알고 있으면 된다.

3. 스택 자료구조

3.1) 스택(Stack) 개요

자료구조(Data Structure)란 '자료를 관리하는 방법'이다. **스택(Stack)**은 자료구조의 일종으로, 자료를 저장하면 가장 최근의 자료부터 빠져나오는 자료구조이다. 책상 위에 접시를 쌓는 경우를 예를 들어보자. 스택에 자료를 넣는다는 것은 1번 접시부터 차례로 2번 접시, 3번 접시를 각각의 접시 위에 올리는 것과 같다. 스택에서 자료를 가져온다는 것은 이렇게 쌓아올린 접시의 가장 위에 있는, 이 경우 3번 접시부터 차례로 한 개씩 접시를 빼내는 것과 같다. 물론 현실에선 접시를 통째로 들어 올리거나, 독특한 사람은 접시를 가운데에서 빼낼 수 있지만, 스택은 그런 식으로 자료를 관리하지 않는다. 가장 나중에 넣은 자료가 가장 먼저 나오게 된다. 다음은 스택의 의사 코드와 그 결과이다.

스택 사용 예제

```
스택.넣는다(1);
스택.넣는다(2);
스택.넣는다(3);
while (스택.비어있는가() == false) { // 스택에 자료가 남아있는 동안
 값 = 스택.꺼낸다();
 출력(값);
}
```

스택 사용 예제 결과

```
3
2
1
```

스택은 다음 행동을 반드시 할 수 있어야 한다.

- 넣는다 : 자료구조는 자료를 관리하는 방법이다. 자료를 넣을 수 없다면 자료구조로 기능할 수 없다.
- 꺼낸다 : 스택은 넣은 자료를 반드시 제거할 수 있어야 한다.

스택에 대해 자료를 넣는 행위를 **푸시(push)**, 자료를 꺼내는 행위를 **팝(pop)**이라고 한다. 위의 의사 코드를 실제로 동작하는 코드로 바꾼다면 다음과 같이 된다.

스택 사용 예제

```
class Stack { /* implementations */ };
int main() {
    Stack stack;
    stack.push(1);
    stack.push(2);
    stack.push(3);
    while (stack.is_empty() == false) {
        int value = stack.pop();
        std::cout << value << std::endl;
    }
    return 0;
}
```

스택을 더 유용하게 만드는 메서드로는 다음과 같은 것이 있다.

- bool is_empty(); // 스택이 비어있는지 확인한다
- Data top(); // 스택에서 가장 최근에 추가된 자료를 확인한다
- int count(); // 스택에 저장된 자료의 수를 가져온다

스택은 일반적으로 두 가지 방법으로 구현한다. 하나는 배열을 이용하는 것이고, 다른 하나는 리스트 자료구조를 이용하는 것이다. JSCC 프로젝트에서는 배열 기반의 스택만 직접 구현해본다.

3.2) 스택 구현

스택을 다음과 같이 설계할 것이다.

- 정수형 자료를 보관하는 스택 클래스를 만든다.
- static const int MAX_STACK_SIZ; // 배열의 크기를 정할 상수
- int list[MAX_STACK_SIZ]; // 정수형 자료를 실제로 보관하는 배열 필드
- int _count; // 스택에 있는 자료의 수를 나타내는 필드
- void push(int data); // 스택에 데이터를 넣는 메서드
- int pop(); // 스택에서 데이터를 꺼내는 메서드
- int top() const; // 스택에 가장 최근에 추가된 데이터를 보는 메서드
- bool is_empty() const; // 스택이 비어있는지 확인하는 메서드
- bool is_full() const; // 스택이 가득 찼는지 확인하는 메서드
- int count() const; // 스택에 있는 자료의 수를 반환하는 메서드

다음은 이 설계를 바탕으로 실제로 구현한 Stack 클래스이다.

01_Stack.cpp

```
typedef std::string Exception; // 임시; 예외로 string 객체를 던진다
class Stack {
    static const int MAX_STACK_SIZ = 10;
    int list[MAX_STACK_SIZ];
    int _count;
public:
    Stack() : _count(0) {} // 스택의 크기를 반드시 0으로 설정해야 한다
    void push(int data) { // 스택에 데이터를 넣는다
```

```
        if (is_full()) { // 스택이 가득차 데이터를 넣을 수 없다면 예외 처리한다
            throw Exception("스택이 가득 찼습니다.");
        }
        // 스택의 마지막에 데이터를 넣고 크기를 증가시킨다
        list[_count++] = data;
    }
    int pop() { // 스택에서 데이터를 꺼낸다
        if (is_empty()) { // 스택이 비어있다면 예외 처리한다
            throw Exception("스택이 비어있습니다.");
        }
        // 스택의 크기를 감소시킨 후 마지막 데이터를 반환한다
        return list[--_count];
    }
    int top() const { // 스택에 가장 최근에 추가된 데이터를 본다
        if (is_empty()) { // 스택이 비어있다면 예외 처리한다
            throw Exception("스택이 비어있습니다.");
        }
        return list[_count - 1]; // 스택의 마지막 데이터를 반환한다
    }
    // 스택이 가득 찼다면 true
    bool is_full() const { return _count == MAX_STACK_SIZ; }
    // 스택이 비어있다면 true
    bool is_empty() const { return _count == 0; }
    // 스택에 저장된 데이터의 수를 가져온다
    int count() const { return _count; }
};
```

이와 같이 스택을 구현하고 이해할 수 있었다.

4. 복합 연산이 가능한 계산기

C언어를 공부했으므로 1+1, 2*8과 같은 단순한 연산이 가능한 계산기를 만들어본 경험이 있으리라 생각한다. 여기서는 복합 연산이 가능한 계산기를 다룬다. 예를 들어 우리가 프로그래밍을 하면서 변수에 값을 대입할 때 우리는 다음과 같은 복잡한 식을 사용할 수 있다.

```
value = is_prime(num) ? a + b * c : d / (e - g) % h;
```

프로그래밍을 어느 정도 공부하고 나면 이러한 연산이 당연히 가능하겠거니 하고 넘어가기 쉽지만, 실제로 이 과정은 생각보다 복잡하고 이해하는 데 노력을 요한다. 여기서는 복합 연산을 이해하고 실제 복합적 연산을 분석하는 계산기를 작성한다. 순서는 다음과 같다.

1. 1+1, 2-4, 6*7, 8/9와 같은 사칙 연산에 대해 동작하는 계산기를 만든다.
2. 식을 넘기면 이를 분석하여 어떤 순서로 연산해야 하는지 출력하는 프로그램을 만든다.
3. 연산 순서를 알고 있고 연산 결과를 알고 있으므로, 이를 이용해 연산한다.

4.1) 사칙 연산 계산기

일단 쉬워 보이는 사칙 연산 계산기부터 작성하자. 내가 공부했던 예제에서는 혼란을 막기 위해 한 자리 수의 정수만으로 연산을 하도록 가정하였으나, 여기서는 정수형의 범위를 벗어나지 않는 임의의 수에 대해 계산이 성립하도록 구현하는 것으로 하겠다. 입력에 대해 다음의 출력이 나오도록 만들자.

입력	출력
4 1+2 3-4 30*40 5	3 -1 1200 5

입력이 5줄인데 결과가 4줄인 건, 맨 위에 입력한 값은 우리가 몇 번 프로그램을 실행하는지를 결정하는 값이기 때문이다. 즉 맨 위에 입력된 4는 프로그램을 4번 수행하겠다는 의미다. 보통 프로그래밍 온라인 저지 사이트에서 자주 사용하는 방법인데, 혹 이에 관심이 생긴다면 '온라인 저지'라고 검색하면 여러 사이트가 나오니 프로그래밍에 자신 있는 사람은 도전해보는 것도 좋다.

뼈대 프로그램을 제시할 테니 빈 칸을 채워 프로그램을 완성하라. 이때 프로그램을 실행하여 식을 입력할 때 사이띄개 하지 않음에 주의하라.

02_basic4.cpp

```
#include <iostream>
typedef std::string Exception;
int calculate(const char *expr); // 넘겨받은 식을 계산하여 값을 반환한다
int main(void) {
    try {
        // 입력의 길이가 MAX_EXPR_LEN보다 큰 경우가
        // 발생하지 않는다고 가정한다
        const int MAX_EXPR_LEN = 256;
        char expression[MAX_EXPR_LEN] = "";
        int loop;
        std::cin >> loop;
        // loop 회수만큼 반복문을 수행한다
        while (loop-- > 0) {
            std::cout << "Enter expression: ";
            std::cin >> expression;
            std::cout << calculate(expression) << std::endl;
        }
        return 0;
    }
    catch (Exception &ex) {
        std::cerr << ex.c_str() << std::endl;
        return 1;
    }
}
int calculate(char expr[]) { // 넘겨받은 식을 계산하여 값을 반환한다
    /* implement it */
}
```

다음은 필자의 구현이다.

02_basic4.cpp

```
int calculate(const char *expr) { // 넘겨받은 식을 계산하여 값을 반환한다
    char ch = *expr;
    if (ch < '0' || '9' < ch) { // 입력의 처음이 숫자가 아니라면 예외 처리
        throw Exception("타당하지 않은 입력입니다.");
    }
    int digit; // 자릿수를 저장할 임시 변수
    // 왼쪽에 나타나는 수 획득
    int left = 0;
    for (ch = *expr; (ch = *expr) != '\0'; ++expr) { // 입력의 끝이 나타나기 전까지
        if (ch < '0' || '9' < ch) { // 수가 아닌 문자가 나타나면 탈출
            break;
        }
        digit = ch - '0'; // 수를 올바른 정수로 바꾼다(문자 '0'은 정수 48과 같다)
        left = 10 * left + digit;
    }
    if (ch == '\0') { // 연산자가 나타나기 전에 입력이 끝났다면
        return left; // 문장의 끝으로 간주하고 획득한 수만 반환
    }
    // 연산자 획득: 사칙 연산에 대해서만 다루므로 연산자 길이는 반드시 1
    char op = *expr++; // 문자열 포인터가 가리키는 연산자를 획득 후 포인터 이동
    // 오른쪽에 나타나는 수 획득: 왼쪽의 경우와 같다
    int right = 0;
    for (ch = *expr; (ch = *expr) != '\0'; ++expr) { // 입력의 끝이 나타나기 전까지
        if (ch < '0' || '9' < ch) { // 수가 아닌 문자가 나타나면 탈출
            break;
        }
        digit = ch - '0'; // 수를 올바른 정수로 바꾼다(문자 '0'은 정수 48과 같다)
        right = 10 * right + digit;
    }
    // 획득한 값과 연산자를 이용하여 연산
    int retVal = 0;
    switch (op) {
    case '+': retVal = left + right; break;
    case '-': retVal = left - right; break;
    case '*': retVal = left * right; break;
    case '/': retVal = left / right; break;
    default: throw Exception("올바른 연산자가 아닙니다.");
    }
    return retVal;
}
```

이와 같이 사칙 연산 계산기를 간단하게 만들 수 있었다.

4.2) 복합 연산 식의 분석

여기서는 복합 연산에 대해 다룬다. 혹시 위의 사칙 연산 문제를 해결하면서 복합 연산식도 간단히 해결할 수 있다고 생각했는가? 그렇다면 직접 프로그램을 만들어보고 다음의 입력에 대해 결과를 잘 출력하는지 확인해보는 것이 좋다.

입력	출력
4 1+2+3 1+2*3+4 3*(2+3) (1+2)*(3+4)+5/6	6 11 15 21

첫 번째 식에 대해서는, 고민하다보면 그저 이전에 연산한 결과를 기록하고 연산자와 피연산자를 찾아 새롭게 연산한 값을 저장하는 행위를 반복하면 된다는 사실을 알 수 있다. 그러면 두 번째 식을 보자. 식의 결과가 13이라고 생각했는가? 그렇다면 계산 과정에서 연산자의 우선순위가 고려되지 않았을 것이다. 사칙연산에선 곱셈과 나눗셈이 덧셈과 뺄셈보다 우선순위가 높으며, 따라서 우리는 2*3을 먼저 계산한 후에 1을 더하고 4를 더해야 한다. 왜 이 경우는 순서가 중요한가? 바로 우선순위가 다른 연산자가 섞여있기 때문이다. 따라서 우리는 식을 분석하는 과정에서 이전에 획득한 연산자의 우선순위와 분석 도중에 새롭게 획득한 연산자의 우선순위를 서로 비교하여, 우선순위가 상대적으로 높은 연산자가 발견된다면 이전의 연산자를 이용한 연산을 일단 뒤로 미룬 다음, 우선순위가 높은 연산자의 연산을 먼저 수행한 다음 처리하지 않았던 연산을 수행해야 한다.

이 과정에서 스택을 이용할 수 있다. 잘 알려진 알고리즘은 다음과 같다.

1. 입력 받은 중위 표기식(infix expression)을 후위 표기식(postfix expression)으로 변환한 후, 변환한 후위 표기식을 해석한다.
2. 중위 표기식을 수식 트리 자료구조로 표현한 후, 수식 트리를 해석한다.
3. 함수의 호출과 반환이 스택과 같으므로, 함수 호출만으로 해석한다.

일반적인 컴파일러는 2번 방법, 즉 수식 트리를 이용하여 식을 표현하고 해석한다. 재귀적 사고에 능숙하다면 3번도 재미있는 해결책이다. 이 문서에서는 1번 방법을 이용하여 먼저 계산기를 구현할 것인데, 그 이유는 이해하기 쉽고 스택 자료구조를 연습하는 데 도움이 되기 때문이다. 하지만 그 전에 알아두어야 하는 개념이 몇 가지 있는데 이를 이해하고 넘어가자.

4.2.1) 식의 표현

수식을 표현하는 방법 중에 연산자의 위치를 기준으로 하는 방법이 있다.

- 전위 표기법(prefix notation) : 연산자가 피연산자 앞에 위치한다. (ex: + 1 2)
- 중위 표기법(infix notation) : 연산자가 피연산자 사이에 위치한다. (ex: 1 + 2)
- 후위 표기법(postfix notation) : 연산자가 피연산자 뒤에 위치한다. (ex: 1 2 +)

이 중에 우리에게 익숙한 표기법은 중위 표기법인데, 위에서 말했지만 연산자의 우선순위가 달라 이대로 분석하기는 어렵다. 그렇다면 후위 표기법은 어떨까? 연산자의 위치가 뒤에 있다는 사실은 알고 있는데 이것만으로 연산자의 우선순위에 관한 문제를 해결할 수 있을까?

먼저 이 후위 표기법이라는 것을 직접 사용해보자. 다음과 같은 중위 표기식이 있다.

```
1 + 2
```

위에서도 예시로 보였지만 이 식을 후위 표기법으로 표현하면 다음과 같다.

```
1 2 +
```

그렇다면 다음은 어떻게 후위 표기법으로 표기할까?

```
1 + 2 + 3
```

이는 각각의 연산을 치환하면 이해하기 쉽다. 먼저 실수 A, B에 대해 다음이 성립한다.

```
A + B = A B +
```

따라서 (1 + 2 = A)로 놓으면 다음이 성립한다.

```
1 + 2 + 3 = A + 3 = A 3 +
```

이때 (A = 1 2 +)이므로 다음이 성립함을 보일 수 있다.

```
1 + 2 + 3 = A 3 + = 1 2 + 3 +
```

그러면 다음 식은 어떨까?

```
1 + 2 * 3
```

먼저 우선순위가 높은 곱셈식부터 문자로 치환해보자. (2 * 3 = A)라고 하면 다음이 성립한다.

```
1 + 2 * 3 = 1 + A = 1 A +
```

이때 (A = 2 3 *)이므로 다음이 성립한다.

```
1 + 2 * 3 = 1 A + = 1 2 3 * +
```

이와 같은 방법으로 다음의 식들을 모두 후위 표기법으로 표기할 수 있다.

```
1 + 2 + 3 = 1 2 + 3 +
1 * 2 + 3 = 1 2 * 3 +
1 + 2 * 3 = 1 2 3 * +
1 * 2 * 3 = 1 2 * 3 *
1 + 2 - 3 + 4 = 1 2 + 3 - 4 +
1 + 2 * 3 + 4 = 1 2 3 * + 4 +
(1 + 2) * (3 - 4) = 1 2 + 3 4 - *
5 * (6 + 7 + 8) - 7 * 9 = 5 6 7 + 8 + * 7 9 * -
```

그러면 이제 변환한 후위 표기식이 잘 변환되었는지 확인해보자. 후위 표기법으로 표현한 식을 다시 중위 표기법으로 표현하는 것이다. 이 또한 전혀 어렵지 않은데, 왜냐하면 방금 우리가 했던 과정을 순서만 바꾸어서 다시 수행하면 되기 때문이다.

위에서 후위 표기법으로 표현된 식을 다시 중위 표기법으로 표현해보자. 왼쪽부터 분석한다.

```
1 2 + 3 +
```

이때 (1 2 + = A)로 놓으면 다음이 성립한다.

```
1 2 + 3 + = A 3 + = A + 3
```

또한 (A = 1 + 2)가 성립하므로 다음이 성립한다.

```
1 2 + 3 + = A + 3 = 1 + 2 + 3
```

이와 같이 역변환이 성립함을 보일 수 있다. 다른 예제를 보자.

```
1 2 3 * +
```

이때 (2 3 * = A)로 놓으면 다음이 성립한다.

```
1 2 3 * + = 1 A + = 1 + A
```

또한 (A = 2 * 3)이 성립하므로 다음이 성립한다.

```
1 2 3 * + = 1 + A = 1 + 2 * 3
```

이와 같이 임의의 식을 후위 표기법, 중위 표기법으로 변환하는 방법을 알아보았다. 그런데 아직 당신은 왜 후위 표기법으로 표현된 식이 중위 표기법으로 표현된 식보다 분석하기 쉬운지에 대한 설명은 듣지 못했다. 정확히 무엇이 더 나아진 것일까?

다음 후위 표기식의 피연산자를 무시하고 연산자만 뜯어서 왼쪽부터 차례로 읽어보라.

```
1 + 2 + 3 = 1 2 + 3 +    : + +
1 * 2 + 3 = 1 2 * 3 +    : * +
1 + 2 * 3 = 1 2 3 * +    : * +
1 * 2 * 3 = 1 2 * 3 *    : * *
1 + 2 - 3 + 4 = 1 2 + 3 - 4 +    : + - +
```

1 + 2 * 3 + 4 = 1 2 3 * + 4 + : * + +
(1 + 2) * (3 - 4) = 1 2 + 3 4 - * : + - *
5 * (6 + 7 + 8) - 7 * 9 = 5 6 7 + 8 + * 7 9 * - : + + * * -

연산자만 뜯어서 왼쪽부터 차례로 살펴보면 아주 재미있는 사실을 발견할 수 있는데, 바로 연산자가 식에서 먼저 연산이 진행되어야 하는 순서로, 즉 높은 우선순위부터 낮은 우선순위로 연산자가 정렬되어 있다는 것이다. 후위 표기법으로 표현된 식에는 연산자의 우선순위에 대한 정보가 이미 포함되어있다. 즉 후위 표기법으로 표현된 식을 분석할 때는 연산자의 우선순위를 고려할 필요가 없다. 바로 이 점 때문에 중위 표기식보다 후위 표기식이 분석하기 수월하다.

이와 같이 후위 표기식이 중위 표기식보다 분석하기 쉬운 이유를 알 수 있었다.

4.2.2) 표기법 변환을 위한 스택의 활용

식을 중위 표기법에서 후위 표기법으로 변환하는 과정은 다음과 같이 진행된다.

1. 피연산자라면 '후위 표기식'의 끝에 추가한다.
2. 연산자라면? -> '연산자 보관소'의 상태를 확인한다.
2.1) '연산자 보관소'에 저장된 연산자가 없다면 추가한다.
2.2) '연산자 보관소'에 저장된 연산자가 있다면? -> 연산자의 우선순위를 비교한다.
2.2.1) 보관소에 마지막으로 저장된 연산자가 우선순위가 높다면 새 연산자보다 우선순위가 낮은 보관소의 모든 연산자를 보관소에서 꺼내 '후위 표기식'의 끝에 추가하고 새 연산자를 보관소에 저장한다.
2.2.2) 새 연산자의 우선순위가 높다면 새 연산자를 보관소에 저장한다.
2.2.3) 두 연산자의 우선순위가 같다면 새 연산자보다 우선순위가 낮은 보관소의 모든 연산자를 보관소에서 꺼내 '후위 표기식'의 끝에 추가하고 새 연산자를 보관소에 저장한다.
3. 1~2 과정을 반복한다.
4. 분석이 끝나면 보관소에 남은 모든 연산자를 '후위 표기식'에 추가한 후 '후위 표기식'을 반환하고 종료한다.

이를 위해 스택을 활용할 수 있다.

1. 피연산자라면 '후위 표기식' 배열의 끝에 추가한다.
2. 연산자라면? -> '연산자 스택'의 상태를 확인한다.
2.2.1) 스택의 최상위 연산자가 우선순위가 높다면 새 연산자보다 우선순위가 낮은 보관소의 모든 연산자를 스택에서 팝 하여 '후위 표기식'의 끝에 추가하고 새 연산자를 스택에 푸시 한다.
2.2.2) 새 연산자의 우선순위가 높다면 새 연산자를 보관소에 푸시 한다.
2.2.3) 두 연산자의 우선순위가 같다면 새 연산자보다 우선순위가 낮은 보관소의 모든 연산자를 스택에서 팝 하여 '후위 표기식'의 끝에 추가하고 새 연산자를 스택에 푸시 한다.
3. 1~2 과정을 반복한다.
4. 분석이 끝나면 스택에 남은 모든 연산자를 '후위 표기식'에 추가한 후 '후위 표기식'을 반환하고 종료한다.

다음은 스택을 활용하여 중위 표기식을 후위 표기식으로 변환하는 예제이다. 소괄호에 대한 처리는 아직 설명하지 않았으므로 적용하지 않았다.

03_in_to_post.cpp

```
#include <iostream>
typedef std::string Exception;
// 공용 함수
inline bool is_digit(char ch) { return ('0' <= ch && ch <= '9'); }
// 스택 정의
typedef char Data;
```

```
class Stack {
    static const int MAX_STACK_SIZ = 256;
    Data _list[MAX_STACK_SIZ];
    int _count;
private:
    inline bool is_full() const { return _count == MAX_STACK_SIZ; }
public:
    Stack() : _count(0) {}
    void push(const Data &data) {
        if (is_full()) throw Exception("Stack is full");
        _list[_count++] = data;
    }
    Data pop() {
        if (is_empty()) throw Exception("Stack is empty");
        return _list[--_count];
    }
    Data top() const {
        if (is_empty()) throw Exception("Stack is empty");
        return _list[_count - 1];
    }
    inline bool is_empty() const { return _count == 0; }
    inline int count() const { return _count; }
};
// 사용할 함수
int op_pri(char ch); // get operator's priority
// convert infix notation expression
//  to postfix notation expression
void infix_to_postfix(char *postfix, const char *infix);
int main(void) {
    try {
        int loop;
        std::cin >> loop;
        const int MAX_EXPR_LEN = 256;
        char infix[MAX_EXPR_LEN] = "";
        char postfix[MAX_EXPR_LEN] = "";
        while (loop-- > 0) {
            std::cout << "Enter expression: ";
            std::cin >> infix;
            infix_to_postfix(postfix, infix);
            std::cout << infix << " : " << postfix << std::endl;
        }
        return 0;
    }
    catch (Exception &ex) {
```

```
        std::cerr << ex.c_str() << std::endl;
        return 1;
    }
}
// 구현
int op_pri(char ch) { // get operator's priority
    int priority = 0;
    switch (ch) {
    case '+': priority = 1; break;
    case '-': priority = 1; break;
    case '*': priority = 2; break;
    case '/': priority = 2; break;
    default: throw Exception("Invalid operator");
    }
    return priority;
}
void infix_to_postfix(char *postfix, const char *infix) {
    // convert infix notation expression
    //  to postfix notation expression
    if (is_digit(*infix) == false)
        throw Exception("Invalid infix notation expression");
    char ch;
    Stack opStack; // operator stack
    for (ch = *infix; (ch = *infix) != '\0'; ++infix) {
        if (is_digit(ch)) {
            while (is_digit(*infix)) {
                *postfix++ = *infix++;
            }
            --infix;
        }
        else { // is_operator
            if (opStack.is_empty() == false) {
                // get operator priority
                int new_pri = op_pri(ch);
                while (opStack.is_empty() == false) {
                    if (new_pri <= op_pri(opStack.top())) {
                        *postfix++ = opStack.pop();
                    }
                    else {
                        break;
                    }
                }
            }
            opStack.push(ch);
```

```
        }
    }
    // 스택에 남은 연산자를
    while (opStack.is_empty() == false) {
        char op = opStack.pop();
        *postfix++ = op;
    }
    *postfix = '\0';
}
```

4.2.3) 소괄호 문제

이제 소괄호를 생각해보자. 다음 중위 표기식을 후위 표기식으로 변환해보라.

```
1 + 2 * 3
(1 + 2) * 3
1 * (2 + 3)
```

답은 다음과 같다.

```
1 + 2 * 3 = 1 2 3 * +
(1 + 2) * 3 = 1 2 + 3 *
1 * (2 + 3) = 1 2 3 + *
```

알고 있듯이, 소괄호는 중위 표기법으로 표현된 식에서는 연산 순서를 강제로 바꾸는 용도로 쓰인다. 따라서 소괄호가 발견된다면, 이전에 어떤 연산자가 존재하든 관계없이 소괄호가 씌워진 식부터 연산을 진행해야 한다. 즉 4.2.2)의 2번 루틴에는 다음 과정이 추가되어야 한다.

1. 여는 소괄호를 만나면**,** 연산자 보관소에 어떤 연산자가 존재하든 관계없이 여는 소괄호를 보관소에 저장한다**.**

2. 닫는 소괄호를 만나면**,** 연산자 보관소에 가장 마지막으로 저장된 여는 소괄호를 발견할 때까지의 모든 연산자를 **‘**후위 표기식**’**에 출력한다**.**

다음은 소괄호 문제가 개선된 표기법 변환 프로그램이다.

04_in_to_post_paren.cpp

```
void infix_to_postfix(char *postfix, const char *infix) {
    // 첫 문자가 수인지 확인하는 코드 삭제
    char ch;
    Stack opStack; // operator stack
    for (ch = *infix; (ch = *infix) != '\0'; ++infix) {
        if (is_digit(ch)) {
            while (is_digit(*infix)) {
                *postfix++ = *infix++;
            }
            --infix; // 반복문 재실행시 ++infix가 실행되므로 한 칸 되돌린다
        }
        else { // is_operator
            if (ch == '(') { // 여는 괄호라면 그냥 넣는다
                opStack.push(ch);
            }
```

```
        else if (ch == ')') { // 닫는 괄호를 발견한 경우의 처리
            if (opStack.is_empty() == false) {
                // get operator priority
                while (opStack.is_empty() == false) {
                    char top = opStack.top();
                    if (top == '(') { // 여는 괄호를 찾았다면 종료
                        break;
                    }
                    else {
                        // 우선순위가 낮은 연산자를 스택에서 꺼내
                        // 후위 표기식에 추가
                        *postfix++ = opStack.pop();
                    }
                }
                // 올바른 괄호 쌍이 존재하는지 확인
                if (opStack.top() != '(') {
                    throw Exception("Invalid parenthesis");
                }
                // 스택에 있는 여는 소괄호를 버린다
                opStack.pop();
            }
        }
        else {
            if (opStack.is_empty() == false) {
                // get operator priority
                int new_pri = op_pri(ch);
                while (opStack.is_empty() == false) {
                    char top = opStack.top();
                    if (top == '(') { // 여는 괄호를 찾았다면 종료
                        break;
                    }
                    else if (new_pri <= op_pri(top)) {
                        *postfix++ = opStack.pop();
                    }
                    else {
                        break;
                    }
                }
            }
            opStack.push(ch);
        }
    }
}
// 스택에 남은 연산자를
```

```
    while (opStack.is_empty() == false) {
        char op = opStack.pop();
        if (op == '(') { // 위에서 처리되지 않은 소괄호가 있다면 예외
            throw Exception("Invalid parenthesis");
        }
        *postfix++ = op;
    }
    *postfix = '\0';
}
```

이와 같이 소괄호 문제를 해결할 수 있었다.

4.2.4) 후위 표기식 분석을 위한 스택의 활용

후위 표기식의 분석은 다음과 같이 진행된다.

1. 피연산자라면 저장한다.
2. 연산자라면 가장 최근에 저장한 두 피연산자와 연산자를 이용하여 연산한 후 다시 저장한다.
3. 1~2 과정을 반복한다.
4. 분석이 끝나면 유일하게 남은 피연산자 하나를 반환하고 종료한다.

이를 위해 스택을 활용할 수 있다.

1. 피연산자 스택을 만들어둔다.
2. 피연산자라면 스택에 푸시 한다.
3. 연산자라면 스택에서 두 피연산자를 팝 하여 연산한 후, 연산 결과를 다시 푸시 한다.
4. 분석이 끝나면 유일하게 스택에 남은 피연산자 하나를 팝하고 종료한다.

스택을 활용하여 후위 표기식을 분석하는 예제를 보이겠다. 이때 입력에 사이띄개 해야 함에 주의하라. 즉 다음과 같이 입력해야 결과가 제대로 나온다.

입력	출력
4 1 2 + 1 2 + 3 + 1 2 3 * + 12 34 +	1 2 + : 3 1 2 + 3 + : 6 1 2 3 * + : 7 12 34 + : 46

스택 클래스의 정의가 템플릿을 이용하여 변경되었음에 유의하라.

05_read_postfix.cpp

```
#include <iostream>
typedef std::string Exception;
// 공용 함수
inline void clear_input_buffer() { // 입력받기 전에 입력 버퍼를 비운다
    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());
}
inline bool is_digit(char ch) { return ('0' <= ch && ch <= '9'); }
```

```
// 문자가 공백인지 확인한다
inline bool is_space(char ch) { return (ch == ' ' || ch == '\t' || ch == '\n'); }
// 스택 정의
template <typename Data> // 형식에 자유로운 스택을 만들기 위해 템플릿 클래스로 변경
class Stack { /* ... */ };
// 사용할 함수
int calculate_postfix(const char *postfix); // calculate postfix expression
int main(void) {
    try {
        int loop;
        std::cin >> loop;
        const int MAX_EXPR_LEN = 256;
        char postfix[MAX_EXPR_LEN] = "";
        while (loop-- > 0) {
            std::cout << "Enter expression: ";
            clear_input_buffer();
            std::cin.getline(postfix, MAX_EXPR_LEN); // 공백을 포함한 줄을 입력받는다
            std::cout << postfix << " : ";
            std::cout << calculate_postfix(postfix) << std::endl;
        }
        return 0;
    }
    catch (Exception &ex) {
        std::cerr << ex.c_str() << std::endl;
        return 1;
    }
}

// 구현
int calculate_postfix(const char *postfix) {
    char ch;
    int digit;
    Stack<int> paramStack;
    for (ch = *postfix; (ch = *postfix) != '\0'; ++postfix) {
        if (is_space(ch)) { // 공백이면 무시한다
            continue;
        }
        else if (is_digit(ch)) { // 피연산자라면 스택에 푸시 한다
            int value = 0;
            while (is_digit(*postfix)) {
                digit = *postfix++ - '0';
                value = 10 * value + digit;
            }
            paramStack.push(value);
```

```
            --postfix; // 반복문 재실행시 ++postfix가 실행되므로 한 칸 되돌린다
        }
        else {
            // 스택에서 두 개의 피연산자를 꺼낸다
            int right = paramStack.pop();
            int left = paramStack.pop();
            int value = 0;
            // 획득한 연산자로 연산한다
            switch (ch) {
            case '+': value = left + right; break;
            case '-': value = left - right; break;
            case '*': value = left * right; break;
            case '/': value = left / right; break;
            default: throw Exception("Invalid operator");
            }
            // 연산 결과를 다시 스택에 넣는다
            paramStack.push(value);
        }
    }
    if (paramStack.count() != 1) { // 스택에 남은 피연산자가 1개가 아니면 예외
        throw Exception("Unhandled operand found");
    }
    return paramStack.pop(); // 하나 남은 피연산자를 반환한다
}
```

이와 같이 후위 표기식을 분석할 수 있었다. 중위 표기식을 후위 표기식으로 변환하는 프로그램, 후위 표기식을 분석하고 계산하는 프로그램을 만들었으므로, 남은 일은 두 프로그램을 하나로 합치는 일이다. 어렵지 않으므로 예제 코드는 이 문서에 싣지 않으나, 혹 궁금할 사람들을 위해 코드는 첨부하겠다.

5. 문자열 버퍼

5.1) 이대로 괜찮을까?

필자가 작성한 코드 중에 아래와 같은, 문자열에서 수를 읽어내는 코드가 있다.

07_conversion.cpp

```
#include <iostream>
// 공용 정의
const int MAX_EXPR_LEN = 256;
typedef std::string Exception;
// 공용 함수
inline void clear_input_buffer() { // 입력받기 전에 입력 버퍼를 비운다
    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());
}
inline bool is_digit(char ch) { return ('0' <= ch && ch <= '9'); }
inline bool is_space(char ch) { return (ch == ' ' || ch == '\t' || ch == '\n'); }
```

```
// 정수와 연산자 사이에 사이띄개를 넣어 출력하는 프로그램
int main(void) {
    try {
        char expression[MAX_EXPR_LEN] = "";
        std::cin >> expression;
        char ch;
        char *expr = expression;
        // 문자열의 모든 문자 탐색
        for (ch = *expr; (ch = *expr) != '\0'; ++expr) {
            if (is_digit(ch)) { // 수가 발견된다면 정수 획득
                int digit = 0; // 자리수를 임시로 저장할 변수
                int value = 0; // 최종적으로 획득할 정수를 저장할 변수
                while (is_digit(*expr)) { // 숫자를 획득하는 동안 value 갱신
                    digit = *expr++ - '0';
                    value = 10 * value + digit;
                }
                --expr; // 반복문 재실행 시 ++expr이 수행되므로 한 칸 되돌린다
                std::cout << value << ' '; // 획득한 정수를 출력하고 뒤에 공백을 추가
            }
            else { // 수가 아닌 문자의 경우 출력하고 공백으로 띄운다
                std::cout << ch << ' ';
            }
        }
        std::cout << std::endl;
        return 0;
    }
    catch (Exception &ex) {
        std::cerr << ex.c_str() << std::endl;
        return 1;
    }
}
```

문제없이 잘 동작한다. 이대로라면 나쁘지 않은 것 같은데? 그렇게 생각한다면 문제를 내겠다. 아래에 수의 제곱, 두 수의 곱, 부피를 구하는 함수가 비어있다. 이를 구현해보라.

08_conversion_skeleton.cpp

```
#include <iostream>
// 공용 정의
const int MAX_EXPR_LEN = 256;
typedef std::string Exception;
// 공용 함수
inline void clear_input_buffer() { // 입력받기 전에 입력 버퍼를 비운다
    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());
}
```

```
inline bool is_digit(char ch) { return ('0' <= ch && ch <= '9'); }
inline bool is_space(char ch) { return (ch == ' ' || ch == '\t' || ch == '\n'); }
// 사용 함수
int get_square(const char *ns); // 문자열을 정수로 변환하고 그 제곱을 반환
// 두 문자열을 정수로 변환하고 그 곱을 반환
int get_mul(const char *ns1, const char *ns2);
int get_volume(const char *x, const char *y, const char *z); // 부피 계산 후 반환
int main(void) {
    try {
        char input1[MAX_EXPR_LEN], input2[MAX_EXPR_LEN], input3[MAX_EXPR_LEN];
        std::cout << "Enter number: ";
        std::cin >> input1;
        std::cout << "Enter number: ";
        std::cin >> input2;
        std::cout << "Enter number: ";
        std::cin >> input3;
        std::cout << get_square(input1) << std::endl;
        std::cout << get_mul(input1, input2) << std::endl;
        std::cout << get_volume(input1, input2, input3) << std::endl;
        return 0;
    }
    catch (Exception &ex) {
        std::cerr << ex.c_str() << std::endl;
        return 1;
    }
}
// 구현
int get_square(const char *ns) { // 문자열을 정수로 변환하고 그 제곱을 반환
    /* implement it */
}
// 두 문자열을 정수로 변환하고 그 곱을 반환
int get_mul(const char *ns1, const char *ns2) {
    /* implement it */
}
int get_volume(const char *xs, const char *ys, const char *zs) { // 부피 계산 후 반
환
    /* implement it */
}
```

필자의 구현은 다음과 같다.

09_conversion_implementation.cpp

```
int get_square(const char *ns) { // 문자열을 정수로 변환하고 그 제곱을 반환
    int digit = 0;
    int value = 0;
```

```
    char ch;
    for (ch = *ns; (ch = *ns) != '\0'; ++ns) {
        if (is_digit(ch) == false) {
            throw Exception("Invalid number");
        }
        digit = ch - '0';
        value = 10 * value + digit;
    }
    return value * value;
}
int get_mul(const char *ns1, const char *ns2) { // 두 문자열을 정수로 변환하고 그 곱을 반환
    int digit = 0;
    int left = 0;
    char ch;
    for (ch = *ns1; (ch = *ns1) != '\0'; ++ns1) {
        if (is_digit(ch) == false) {
            throw Exception("Invalid number");
        }
        digit = ch - '0';
        left = 10 * left + digit;
    }
    int right = 0;
    for (ch = *ns2; (ch = *ns2) != '\0'; ++ns2) {
        if (is_digit(ch) == false) {
            throw Exception("Invalid number");
        }
        digit = ch - '0';
        right = 10 * right + digit;
    }
    return left * right;
}
int get_volume(const char *xs, const char *ys, const char *zs) { // 부피 계산 후 반환
    int digit = 0;
    int x = 0;
    char ch;
    for (ch = *xs; (ch = *xs) != '\0'; ++xs) {
        if (is_digit(ch) == false) {
            throw Exception("Invalid number");
        }
        digit = ch - '0';
        x = 10 * x + digit;
    }
```

```
    int y = 0;
    for (ch = *ys; (ch = *ys) != '\0'; ++ys) {
        if (is_digit(ch) == false) {
            throw Exception("Invalid number");
        }
        digit = ch - '0';
        y = 10 * y + digit;
    }
    int z = 0;
    for (ch = *zs; (ch = *zs) != '\0'; ++zs) {
        if (is_digit(ch) == false) {
            throw Exception("Invalid number");
        }
        digit = ch - '0';
        z = 10 * z + digit;
    }
    return x * y * z;
}
```

무엇이 문제인지 알겠는가? 바로 문자열을 정수로 변환하는 코드가 지나치게 중복적이라는 것이다. 알고 있겠지만, 이렇게 코드를 작성하면 가독성이 떨어지고, 코드 하나가 잘못되었거나 후에 업데이트해야 하는 경우, 이렇게 작성한 모든 코드를 전부 찾아내 수정해야 하므로 유지 및 보수에 애로사항이 꽃피게 된다. 어떻게 해결해야할까?

여기서 우리는 atoi 함수처럼, 문자열에서 정수를 긁어내는 함수를 작성해볼 수 있다.

10_atoi.cpp

```
int ascii_to_int(const char *ns) { // 문자열을 정수로 변환하고 그 값을 반환
    // 중요: 정수가 아닌 문자가 나타날 때까지 분석을 진행한다
    int digit = 0;
    int value = 0;
    char ch;
    for (ch = *ns; (ch = *ns) != '\0'; ++ns) {
        if (is_digit(ch) == false) {
            break;
        }
        digit = ch - '0';
        value = 10 * value + digit;
    }
    return value;
}
int get_square(const char *ns) { // 문자열을 정수로 변환하고 그 제곱을 반환
    int value = ascii_to_int(ns);
    return value * value;
}
```

```
int get_mul(const char *ns1, const char *ns2) { // 두 문자열을 정수로 변환하고 그 곱을 반환
    int left = ascii_to_int(ns1), right = ascii_to_int(ns2);
    return left * right;
}
int get_volume(const char *xs, const char *ys, const char *zs) { // 부피 계산 후 반환
    return ascii_to_int(xs) * ascii_to_int(ys) * ascii_to_int(zs);
}
```

세 함수의 코드 길이가 크게 줄었음을 볼 수 있다. 또한 함수 이름이 함수가 하는 내용을 암시하고 있으므로, 함수 내부를 들여다보았을 때 내부적으로 어떻게 동작하는지를 이해하기도 쉽다. 이렇듯 문자열로부터 정수를 긁어내는 작업은 코드 길이가 길고 반복적으로 사용되기 때문에 반드시 뜯어내야 한다.

정수를 획득하는 코드를 뜯어내야 한다는 사실을 알았으니, 이것이 적용되지 않은 이전의 예제들도 수정해야 한다는 생각이 든다. 옳은 판단이고 지금 바로 예제를 수정해보려 한다. 초반에 작성했던 사칙연산 계산기 예제부터 수정해볼까? 다음은 atoi를 이용하여 사칙 연산을 개선한 예제다.

11_basic4_atoi.cpp

```
int calculate(const char *expr) { // 넘겨받은 식을 계산하여 값을 반환한다
    if (is_digit(*expr) == false) { // 입력의 처음이 숫자가 아니라면 예외 처리
        throw Exception("타당하지 않은 입력입니다.");
    }
    int left = ascii_to_int(expr); // 왼쪽에 나타나는 수 획득
    while (is_digit(*expr)) { ++expr; } // 정수가 아닐 때까지 expr를 이동한다
    if (*expr == '\0') { // 입력이 끝났다면 획득한 정수를 반환하고 종료
        return left;
    }
    // 연산자 획득: 사칙 연산에 대해서만 다루므로 연산자 길이는 반드시 1
    char op = *expr++; // 문자열 포인터가 가리키는 연산자를 획득 후 포인터 이동
    int right = ascii_to_int(expr); // 오른쪽에 나타나는 수 획득
    while (is_digit(*expr)) { ++expr; } // 정수가 아닐 때까지 expr를 이동한다
    if (*expr != '\0') { // 입력이 아직 끝나지 않았다면 예외 발생
        throw Exception("두 개의 피연산자로만 연산할 수 있습니다.");
    }
    // 획득한 값과 연산자를 이용하여 연산
    int retVal = 0;
    switch (op) {
    case '+': retVal = left + right; break;
    case '-': retVal = left - right; break;
    case '*': retVal = left * right; break;
    case '/': retVal = left / right; break;
    default: throw Exception("올바른 연산자가 아닙니다.");
    }
    return retVal;
```

```
}
```

예제를 유심히 보면 이상한 부분이 있다. ascii_to_int 함수는 넘긴 문자열로부터 정수를 읽는 함수다. 즉 이미 정수를 읽었는데도 불구하고 문자열 포인터에서 다시 정수를 확인하며 지나기고 있다. 왜 같은 행위를 두 번이나 하는가? 우리가 이전에 작성했던 예제에서는 정수를 읽고 나면 포인터가 이동해있었는데, 왜 이 예제에서는 그렇지 않고 우리가 다시 포인터를 옮겨주어야 하는가?

그 이유는 아주 당연한데, 함수 호출 시에 인자를 넘길 때는 언제나 넘기려는 값의 사본이 넘어가기 때문이다. 포인터를 넘긴다면 포인터가 저장하는 값의 사본이, 참조를 넘긴다면 참조의 위치 값의 사본이 넘어간다. 위 예제에서는 expr 포인터를 넘겼으니 expr 포인터가 가리키는 문자 배열의 주소 값을 사본으로 전달했지만, 이것이 expr 포인터 변수의 주소는 아니라는 것이다. ns는 atoi 함수에서 정의된 지역 변수이고, expr은 calculate 함수에서 정의된 지역 변수이다. 즉 두 변수에 대해 (ns == expr)은 성립하지만 (&ns == &expr)는 성립하지 않는다. expr의 값을 수정하려면 &expr 또한 함수의 인자로 넘겨야만 한다.

이 문제를 해결하려면 atoi의 인자로 expr의 주소도 같이 넘기던가, 아니면 atoi의 인자를 expr의 참조로 하던가 해야 한다. 결국 다음과 같은 식인데 어느 쪽도 그렇게 깔끔하지 않다.

12_atoi_alt.cpp

```
int ascii_to_int_adr(const char **ns_src) { // 문자열 포인터의 주소를 넘긴다
    const char *ns = *ns_src; // 문자열 포인터가 가리키는 값을 획득한다
    int digit = 0;
    int value = 0;
    char ch;
    for (ch = *ns; (ch = *ns) != '\0'; ++ns) {
        if (is_digit(ch) == false) {
            break;
        }
        digit = ch - '0';
        value = 10 * value + digit;
    }
    *ns_src = ns; // 분석이 끝나면 문자열 포인터 변수에 값을 대입한다
    return value;
}
int ascii_to_int_ref(const char *&ns) { // 문자열 포인터의 참조를 넘긴다
    int digit = 0;
    int value = 0;
    char ch;
    for (ch = *ns; (ch = *ns) != '\0'; ++ns) {
        if (is_digit(ch) == false) {
            break;
        }
        digit = ch - '0';
        value = 10 * value + digit;
    }
    return value;
}
```

```
int calculate(const char *expr) { // 넘겨받은 식을 계산하여 값을 반환한다
    if (is_digit(*expr) == false) { // 입력의 처음이 숫자가 아니라면 예외 처리
        throw Exception("타당하지 않은 입력입니다.");
    }
    int left = ascii_to_int_adr(&expr); // 왼쪽에 나타나는 수 획득 (주소)
    if (*expr == '\0') { // 입력이 끝났다면 획득한 정수를 반환하고 종료
        return left;
    }
    // 연산자 획득: 사칙 연산에 대해서만 다루므로 연산자 길이는 반드시 1
    char op = *expr++; // 문자열 포인터가 가리키는 연산자를 획득 후 포인터 이동
    int right = ascii_to_int_ref(expr); // 오른쪽에 나타나는 수 획득 (참조)
    if (*expr != '\0') { // 입력이 아직 끝나지 않았다면 예외 발생
        throw Exception("두 개의 피연산자로만 연산할 수 있습니다.");
    }
    // 획득한 값과 연산자를 이용하여 연산
    int retVal = 0;
    switch (op) {
    case '+': retVal = left + right; break;
    case '-': retVal = left - right; break;
    case '*': retVal = left * right; break;
    case '/': retVal = left / right; break;
    default: throw Exception("올바른 연산자가 아닙니다.");
    }
    return retVal;
}
```

필자는 이에 대한 대안으로 문자열 버퍼 클래스 StringBuffer를 제안한다.

5.2.2) StringBuffer 클래스

StringBuffer 클래스는 문자열을 다루기 쉽도록 필자가 고안한 클래스다. StringBuffer 클래스의 인스턴스는 다음의 행위를 수행할 수 있다.

- init: 버퍼를 인자로 받은 문자열로 초기화한다
- getc: 버퍼로부터 문자를 하나 가져온다
- ungetc: 버퍼에서 읽었던 값을 되돌린다
- add: 버퍼의 끝에 문자 또는 문자열을 추가한다
- is_empty: 버퍼가 비어있는지 확인한다

다음은 StringBuffer 클래스를 구현한 것이다.

StringBuffer.h

```
#ifndef __HANDY_STRINGBUFFER_H__
#define __HANDY_STRINGBUFFER_H__

#include <string>
class StringBuffer {
    std::string str;
```

```
    unsigned len;
    unsigned idx;

public:
    StringBuffer(const char *s = "");
    StringBuffer(const std::string &str);
    ~StringBuffer();

    // 버퍼를 문자열로 초기화합니다.
    void init(const char *str);
    void init(const std::string &str);

    // 버퍼로부터 문자를 하나 읽습니다. 포인터가 이동합니다.
    char getc();
    // 버퍼의 포인터가 가리키는 문자를 가져옵니다. 포인터는 이동하지 않습니다.
    char peekc() const;
    // 버퍼에서 읽었던 값을 되돌립니다. 되돌릴 수 없으면 false를 반환합니다.
    bool ungetc();

    // 버퍼의 끝에 문자 또는 문자열을 추가합니다.
    void add(char c);
    void add(const char *s);
    void add(const std::string &str);

    // 버퍼가 비어있다면 true, 값을 더 읽을 수 있다면 false를 반환합니다.
    bool is_empty() const;
};

#endif
```

StringBuffer.cpp

```
#include "StringBuffer.h"
#include <string>
typedef std::string Exception;
StringBuffer::StringBuffer(const char *s) : str(s), idx(0) { this->len =
this->str.length(); }
StringBuffer::StringBuffer(const std::string &str) : str(str), idx(0) { this->len
= this->str.length(); }
StringBuffer::~StringBuffer() {}
void StringBuffer::init(const char *str) {
    this->str = str;
    this->idx = 0;
    this->len = this->str.length();
}
```

```
void StringBuffer::init(const std::string &str) {
    this->str = str;
    this->idx = 0;
    this->len = this->str.length();
}
char StringBuffer::getc() {
    if (idx >= len) {
        throw Exception("Buffer is empty");
    }
    return str[idx++];
}
char StringBuffer::peekc() const {
    if (idx >= len) {
        throw Exception("Buffer is empty");
    }
    return str[idx];
}
bool StringBuffer::ungetc() {
    if (idx > 0) {
        --idx;
        return true;
    }
    else {
        return false;
    }
}
void StringBuffer::add(char c) {
    this->str += c;
}
void StringBuffer::add(const char *s) {
    this->str += s;
}
void StringBuffer::add(const std::string &str) {
    this->str += str;
}
bool StringBuffer::is_empty() const {
    return (idx >= len);
}
```

다음은 StringBuffer 클래스를 사용하는 예제이다.

13_StringBufferMain.cpp

```
#include <iostream>
#include "StringBuffer.h"
typedef std::string Exception;
```

```
int main(void) {
    try {
        StringBuffer buffer;
        buffer.init("Hello, world!");
        while (buffer.is_empty() == false) {
            std::cout << buffer.getc() << std::endl;
        }
        return 0;
    }
    catch (Exception &ex) {
        std::cerr << ex.c_str() << std::endl;
        return 1;
    }
}
```

그리고 이를 이용하여 사칙 연산 계산기를 다시 작성할 수 있다.

14_basic4_StringBuffer.cpp

```
int ascii_to_int(StringBuffer &buffer) { // 문자열을 정수로 변환하고 그 값을 반환
    int digit = 0;
    int value = 0;
    char ch;
    while (buffer.is_empty() == false) {
        if (is_digit(ch = buffer.getc()) == false) {
            buffer.ungetc(); // 아직 읽지 않은 값이므로 되돌린 후 탈출한다
            break;
        }
        digit = ch - '0';
        value = 10 * value + digit;
    }
    return value;
}
int calculate(const char *expr) { // 넘겨받은 식을 계산하여 값을 반환한다
    StringBuffer buffer(expr);
    if (is_digit(buffer.peekc()) == false) { // 입력의 처음이 숫자가 아니라면 예외
        throw Exception("타당하지 않은 입력입니다.");
    }
    int left = ascii_to_int(buffer); // 왼쪽에 나타나는 수 획득
    if (buffer.is_empty()) { // 입력이 끝났다면 획득한 정수를 반환하고 종료
        return left;
    }
    // 연산자 획득: 사칙 연산에 대해서만 다루므로 연산자 길이는 반드시 1
    char op = buffer.getc(); // 문자열 포인터가 가리키는 연산자를 획득 후 포인터 이동
    int right = ascii_to_int(buffer); // 오른쪽에 나타나는 수 획득
    if (buffer.is_empty() == false) { // 입력이 아직 끝나지 않았다면 예외 발생
```

```
        throw Exception("두 개의 피연산자로만 연산할 수 있습니다.");
    }
    // 획득한 값과 연산자를 이용하여 연산
    int retVal = 0;
    switch (op) {
    case '+': retVal = left + right; break;
    case '-': retVal = left - right; break;
    case '*': retVal = left * right; break;
    case '/': retVal = left / right; break;
    default: throw Exception("올바른 연산자가 아닙니다.");
    }
    return retVal;
}
```

독자 중엔 StringBuffer 클래스가 이전과 크게 편의성에서 차이가 나지 않는다고 생각하는 사람도 있을 것이다. 그 경우에는 자신이 원하는 방법으로 진행하면 된다. 사실 StringBuffer 클래스는 완성된 클래스가 아니고, 프로젝트를 진행하면서 기능을 추가하여, 이 클래스가 없이 작업하는 것을 상상할 수 없을 정도로 만들 것이다. 이와 같이 StringBuffer 클래스를 활용하고 이해할 수 있었다.

6. 단원 마무리

원래는 1단원의 마지막을 변수 처리가 가능한 복합 연산 계산기로 하려고 했는데, 아무래도 1단원에서 설명하기 지나치게 복잡한 감이 없지 않아 있어서 뒤로 미루었다. 스택에 대해 설명하였으나 이전에 설명해본 적이 있는 것도 그림이 첨부되어있는 것도 아니어서, 스택이라는 자료구조를 처음 접한 사람에게는 상당히 어려운 시작이었을 거라는 생각이 들어 안타깝다. 복합 연산 계산기 또한 스택을 이미 알고 있는 사람이라도 헷갈리기 쉬운 내용이라 설명이 잘 되었을지 걱정이 앞선다. 여기까지 따라왔다면 정말 열심히 이 문서를 읽은 것이고 칭찬 받아야 마땅하다고 생각한다(내 글을 읽은 걸 칭찬해야 한다고 쓰니 웃기는 일이지만). 아직 이해가 되지 않았다면 '윤성우 저 열혈 자료구조'를 참고하면 좋을 것이라 생각한다. 쉬운 설명과 그림이 곁들여져있는 아주 좋은 책이다.