본 PSet 은 저의 강의 경험과 학생들의 의견 및 Stanford CS106 과 Harvard CS50 같은 강의에서 수집된 자료를 토대로 작성되었습니다. 본 PSet 에 문제가 있거나, 질문 혹은 의견이 있다면, 언제든지 알려 주시면 감사하겠습니다. 강의 개선에 많은 도움이 되겠습니다.
idebtor@gmail.com

PSet: Infix & Postfix Evaluations

목차

과제 수행 목적

본 프로젝트의 목적

- postfix, infix, stack 등의 핵심 알고리즘 학습
- c++ 템플릿과 c++ STL 의 multiple stacks 사용 경험
- visual studio 와 디버깅 사용 경험 (매우 중요!)

제공되는 파일 목록

- infixpostfix.pdf - 본 파일
- postfixDriver.cpp - 수정 금지, 제출 불필요
- postfix.cpp - 뼈대 코드
- postfix.exe - 참고용 실행 파일
- infix.cpp - 뼈대 코드
  infixall.cpp - 참고용 파일이므로 infix.cpp 를 사용하세요.
  infix.cpp 작성을 완료한 후 infixall.cpp 에 복사 및 붙여넣기 하여 사용하세요.
- infixDriver.cpp - 수정 금지, 제출 불필요
- infixallDriver.cpp - 수정 금지, 제출 불필요
- infixallx.exe - 참고용 실행 파일

Mac 사용자: Windows 용 실행 파일(.exe)을 실행하기 위해 Wine 혹은 WineBottler 를 사용하세요.

개요

목표 1: postfix 식을 infix 식으로 계산하기
목표 2: 문자열로 이루어진 infix 산술식을 계산하고 결괏값 출력하기
- 이 식은 괄호를 포함할 수 있으며 괄호가 서로 일치해야 합니다.
- 편의를 위해 2 진 연산(+, -, *, /)만 허용해도 됩니다.

- **Infix 표기법**: 연산자가 피연산자 사이에 위치 (예. 3 + 4)
- **Prefix 표기법**: 연산자가 피연산자 이전에 위치 (예. + 3 4)
- **Postfix 표기법**: 연산자가 피연산자 이후에 위치 (예. 3 4 +)

Step 1: postfix.cpp & postfixDriver.cpp

1 단계는 2 개의 파트로 구성되어 있습니다.
첫 번째 파트는 postfix 식을 완벽히 괄호로 묶은(fully parenthesized) infix 식으로 나타내는 함수를 구현하는 것입니다. 이를 위해 다음 함수들을 완성해야 합니다.

```
string evaluate(string tokens);
void printStack(stack<T> orig);  // a helper function, use function template
```

첫 번째 파트를 완성한 후, 아래에 나온 함수들을 구현하세요.

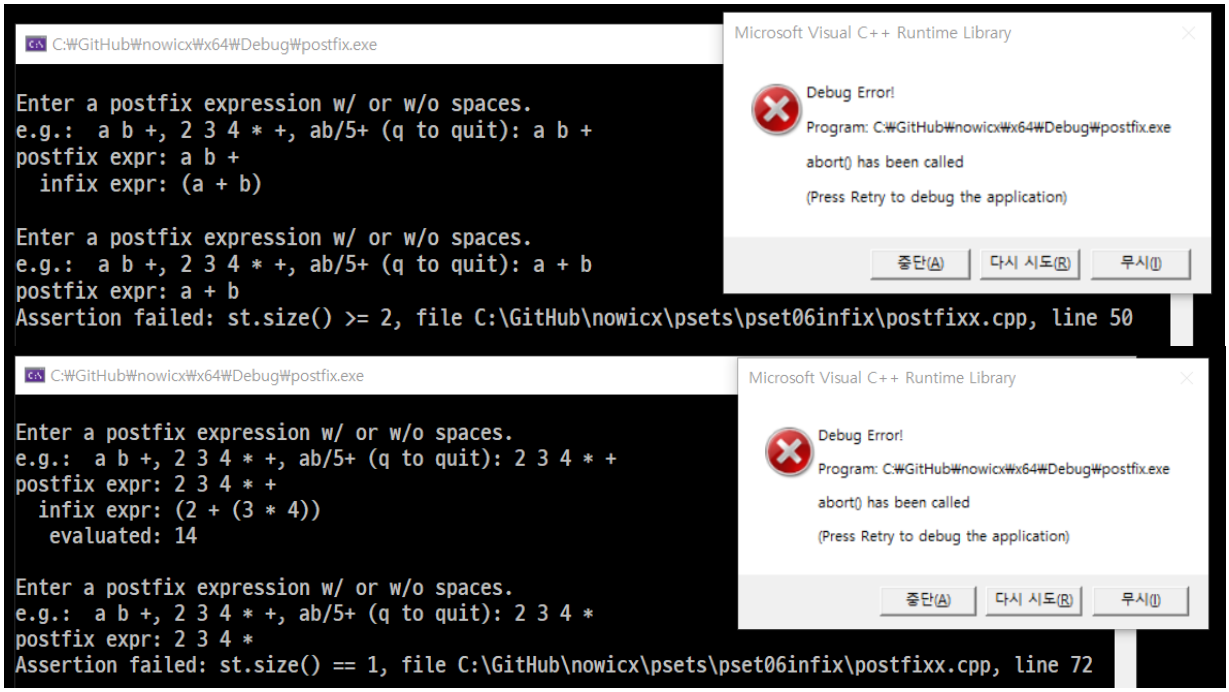
```
bool is_numeric(string tokens);
double evaluate_numeric(string tokens);
```

- **is_numeric():** postfix 식이 숫자로 표현되어 있다면 'true'을 반환합니다. 주어진 postfix 식이 숫자와 연산자로만 구성되어 있다면 **evaluate_numeric()** 을 호출합니다.
- **evaluate_numeric()**: **evaulate()** 함수와 비슷합니다. 차이점은 산술 연산의 결괏값(숫자 값)이 stack 에 저장되므로 stack<string> 대신 **stack<double>**을 사용합니다.
소스 파일에 제공된 스켈레톤 코드의 주석과 다음 지침을 따라 코드를 완성하세요.

- **evaluate()** 함수는 postfix 식을 계산한 후, 완벽히 괄호로 묶은 infix 식을 string 형태로 반환합니다. 계산에는 **stack<string>**을 사용합니다.
- **evaluate_numeric()** 함수는 postfix 식을 계산하여 숫자 값을 반환합니다. 계산에는 **stack<double>**을 사용합니다.
- **is_numeric()** 함수는 postfix 식이 기호가 아닌 숫자로 표현되어 있다면 '참'을 반환합니다.
- C++ STL stack 클래스를 사용하여 **printStack(stack<T> st)**을 구현하세요. 이 함수는 stack 의 가장 아래에 위치한 요소부터 차례대로 출력하며, 디버깅에 용이합니다.
- 코딩의 간소화를 위해 postfix 식은 단일 문자 피연산자와 연산자만으로 구성되어 있으며 공백을 포함하기도 합니다.
- **evaluate()**에서 stack 의 크기를 확인하기 위해 **assert()** macro 를 적절한 곳에 사용하세요. 조건이 참일 경우, 프로그램은 정상적으로 실행됩니다. 조건이 거짓일 경우, 프로그램이 종료되고 에러 메시지가 나타납니다. assert()의 일반적인 구문은 다음과 같습니다.

```
assert (bool_conditional_expression)
```

실행 예시:

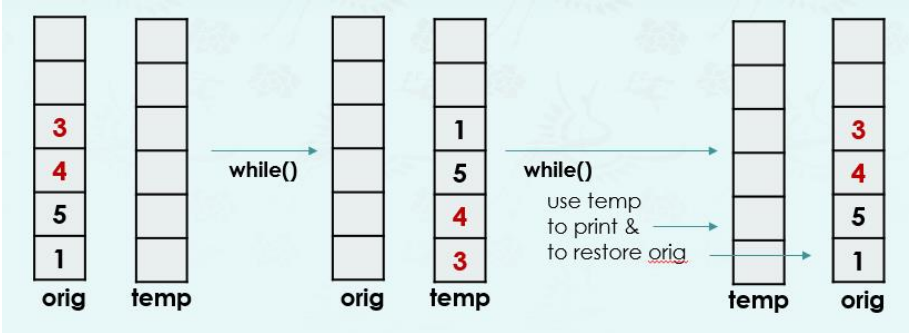


printStack()

이 PSet 을 진행하는 동안 stack 의 현재 상태를 알고자 합니다. stack 의 요소를 아래에서부터 위로 출력하는 도우미 함수를 작성하세요. 이전에 재귀(recursion) 알고리즘을 사용하여 구현한 적이 있으므로 이번에는 반복(iteration)을 사용하여 구현하세요.
이번 단계에서 다음과 같이 **printStack(stack<T> st)**을 구현하세요. C++/STL 이 제공하는 stack 클래스를 사용할 수 있습니다. 이후 단계에서 recursion 을 이용하여 이 함수를 구현할 것입니다.

알고리즘:

- **orig** 이라는 이름의 stack 이 제공됩니다.
- **temp** 라는 이름의 빈 stack 을 생성하세요.
- **orig** 이 빈 stack 이 될 때까지
  - **orig** 의 요소를 Top()/Pop()하여 **temp** 로 push()하세요.
- **temp** 가 빈 stack 이 될 때까지
  - **temp** 의 요소를 Top()/Pop()하여 출력한 후, **orig** 로 push()하세요.

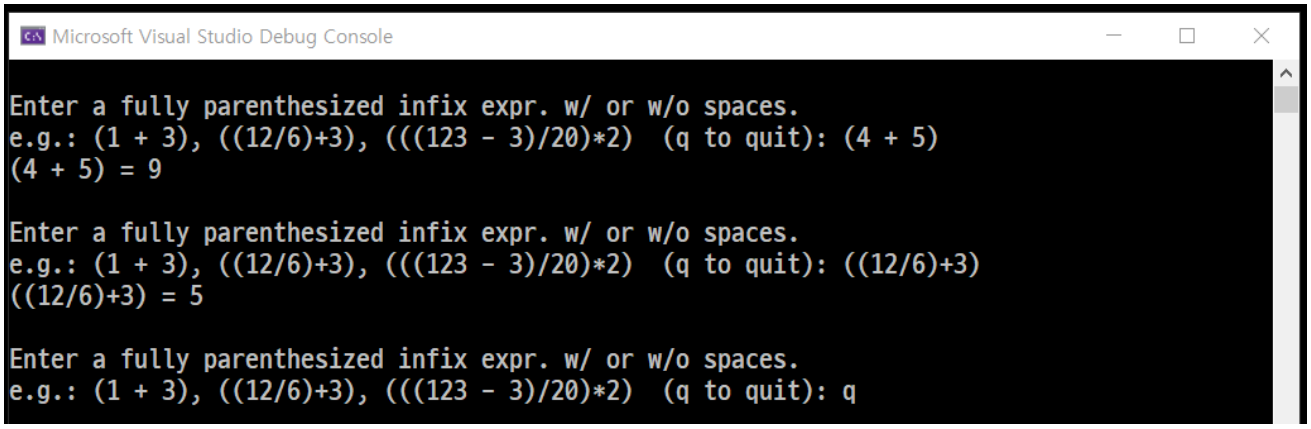


.

Step 2: infix 산술식 계산하기

두 번째 파트는 infix 식을 계산하는 것입니다. 사용자가 공백을 포함하거나 포함하지 않은 infix 식을 입력하면 코드가 실행되며 결과가 출력됩니다. Infix 식은 우선순위(precedence) 결정에 추가 작업이 필요하여 컴퓨터가 계산하기 더 어렵습니다.
Edgar Dijkstra 가 제안한 알고리즘으로, 2 개의 stack 을 사용하여 infix 표기법을 계산하는 유명한 알고리즘이 있습니다. Dijkstra 의 아이디어는 1 개의 stack 을 사용하는 대신, 하나는 피연산자, 하나는 연산자를 위한 2 개의 stack 을 사용하는 것이었습니다.

실행 예시:



Operator stack 과 Operand stack

우리가 코딩할 이 프로그램은 infix 식을 string 형태로 받아 결과를 반환합니다. 식의 각 연산자(Operator)는 단일 문자로, 각 피연산자(Operand)는 정숫값으로 표현됩니다. 숫자는 여러 자리일 수 있습니다. 이 과정에서 하나는 연산자, 다른 하나는 피연산자를 위한 2 개의 stack 을 사용합니다.

- operand stack: 이 stack 은 숫자를 저장하는 데 사용됩니다.
- operator stack: 이 stack 은 연산 작업(+, -, *, /)을 위해 사용됩니다.

이전 단계에서 구현한 printStack(stack<T>를 사용하세요. 이번에는 이 함수가 stack<int>와 stack<char> 데이터 유형 둘 다에 사용됩니다.

Infix 산술식 예시:

Infix 산술식 구성 예시:
1 - 3
2 * ((3 - 7) + 46)
(12 + (4 * 100)) - (2* 5)
(((2 + 4) * 100) - (2 *5)) + 1
12 + 4 * 100 - 2* 5
(2 + 4) * 100 - 2 *5 + 1

- 숫자:
  모든 숫자는 내부에서 정숫값으로 표현됩니다. 지수는 정수의 분할로 수행됩니다.
- 괄호:
  일반적으로 사용되는 의미와 같습니다. ( )만 사용 가능하며, {} []는 사용하지 마세요.
- 연산자:
  - + 덧셈, 이진 연산자로만 사용
  - - 뺄셈, 이진 연산자로만 사용
  - * 곱셈
  - / 나눗셈

코딩:

infix.cpp 에 제공된 뼈대 코드는 부분적으로 작동할 것입니다. 다음 항목들을 구현해서 코드를 완성하세요.

1. Postfix.cpp 에서 도우미 함수 **printStack(stack<T>)** 을 복사해오세요.
2. 여러 자리의 정숫값(피연산자)을 허용하세요.
3. **compute()** 함수에 존재하는 버그를 수정하세요.
4. **assert()** macro 를 적절한 곳에 사용하여 stack 의 크기를 확인하세요.
5. "Your code here" 부분에 코드를 작성하세요. 대부분 **evaluate()**에 존재합니다**.**

코딩의 간소화를 위해 식이 **처음과 마지막을 제외하고 완전히 괄호로 묶여 있다**고 가정합니다. 이 방식은 코드 구현에 연산자의 우선순위를 필요로 하지 않지만, 사용자가 연산에 필요한 모든 괄호를 반드시 정확하게 입력해야 합니다. 예를 들어, 다음 식은 완성된 코드로 올바르게 작동하고 정확한 결과를 출력해야 합니다.

1 - 3 = -2
((3 - 1) * 5) - 4 = 6
1 + (234 - 5) = 230
123 - (21 * 5) = 18
(12 - 8) * (45 / 3) = 60

Infix 식을 계산하기 위한 대략적인 알고리즘은 다음과 같습니다.

1 읽어올 토큰이 없을 때까지
  1.1 다음 토큰을 가져옵니다.
  1.2 만약 토큰이
    1.2.1 공백일 경우, 무시합니다.
    1.2.2 왼쪽 괄호일 경우, 무시합니다.
    1.2.3 숫자일 경우
      1.2.3.1 숫자를 읽어옵니다 (여러 자리 수일 수 있습니다).
      1.2.3.2 숫자를 value stack 에 push()합니다.
    1.2.4 오른쪽 괄호일 경우
      1.2.4.1 operator stack 에서 연산자를 pop()합니다.
      1.2.4.2 operand stack 을 2 번 pop()하여 2 개의 피연산자를 가져옵니다.
      1.2.4.3 연산자를 올바른 순서로 피연산자에 적용합니다.

1.2.4.4 결괏값을 operand stack 에 push()합니다.
1.2.5 연산자일 경우
1.2.5.1 operator stack 에 연산자를 push()합니다.

2 (전체 식을 모두 분석했으니 Op stack 에 남아있는 연산자들을 operand stack 에 남아있는 피연산자들에 적용합니다)
operator stack 이 비워질 때까지
2.1 operator stack 에서 연산자를 pop()합니다.
2.2 operand stack 을 2 번 pop()하여 2 개의 피연산자를 가져옵니다.
2.3 연산자를 올바른 순서로 피연산자에 적용합니다.
2.4 결괏값을 operand stack 에 push()합니다.

3 (이 시점에서 operator stack 은 비어 있어야 하며, operand stack 에는 단 1 개의 결괏값만 저장되어 있어야 합니다.)
operand stack 의 맨 위 요소를 반환합니다.

Step 3: infixall.cpp

2 단계의 모든 기능 구현을 마쳤다면 **infix.cpp** 를 **infixall.cpp** 에 복사하여 붙여 넣습니다. 다음 내용을 코드로 작성합니다.

- 지수 연산자 ^를 추가합니다. 예를 들어, 1+2^3+2 는 11 을 반환합니다.
- Infix 식에서 **"완벽히 괄호로 묶은"이라는 제약을 제거**하여 코드를 개선합니다.
- 연산자에 따라 0, 1, 2 ...를 반환하는 **precedence()** 함수를 추가합니다.
- **재귀를 사용하여** Template 버전의 printStack()이 자신이 정의한 stack 이 아닌 시스템 stack 을 사용하도록 다시 작성하세요.
  사용자의 stack 에서 요소를 top()/pop()하고 stack 의 맨 밑에 도달할 때까지 재귀 함수 printStack()을 호출합니다. 재귀 호출을 하는 동안, top()/pop()한 요소들은 **시스템 stack** 에 자동으로 저장됩니다. 사용자 stack 이 비어지면, 마지막으로 pop()한 요소를 출력하기 시작합니다. 이렇게 하면 맨 아래 요소부터 맨 위 요소까지 차례대로 출력될 것입니다. 그런 다음 출력된 요소를 다시 집어넣으면, stack 에서 요소의 순서가 유지됩니다. 의사 코드(pseudo code)는 다음과 같습니다.

```
void printStack(stack<T> st)
    if stack is empty, return
    get the item at top from stack and remove it from the original stack
    printStack()      // recursive calls until it reaches the bottom of stack
    print the item    // recursive calls ended, recover item from system stack
    push the item     // reconstruct the original stack
```

예를 들어, 다음 식은 완성된 코드로 올바르게 작동하고 정확한 결과를 출력해야 합니다.
(1 + 2^ 3) * 5-1 = 44
2*(21-6)/5 = 6
2 * (3 - 7 + 46) = 84
(2 + 4) * 100 - 2 *5 + 1= 591

대략적인 알고리즘은 다음과 같습니다. 에러 검사는 따로 하지 않으니 직접 추가해야 합니다.

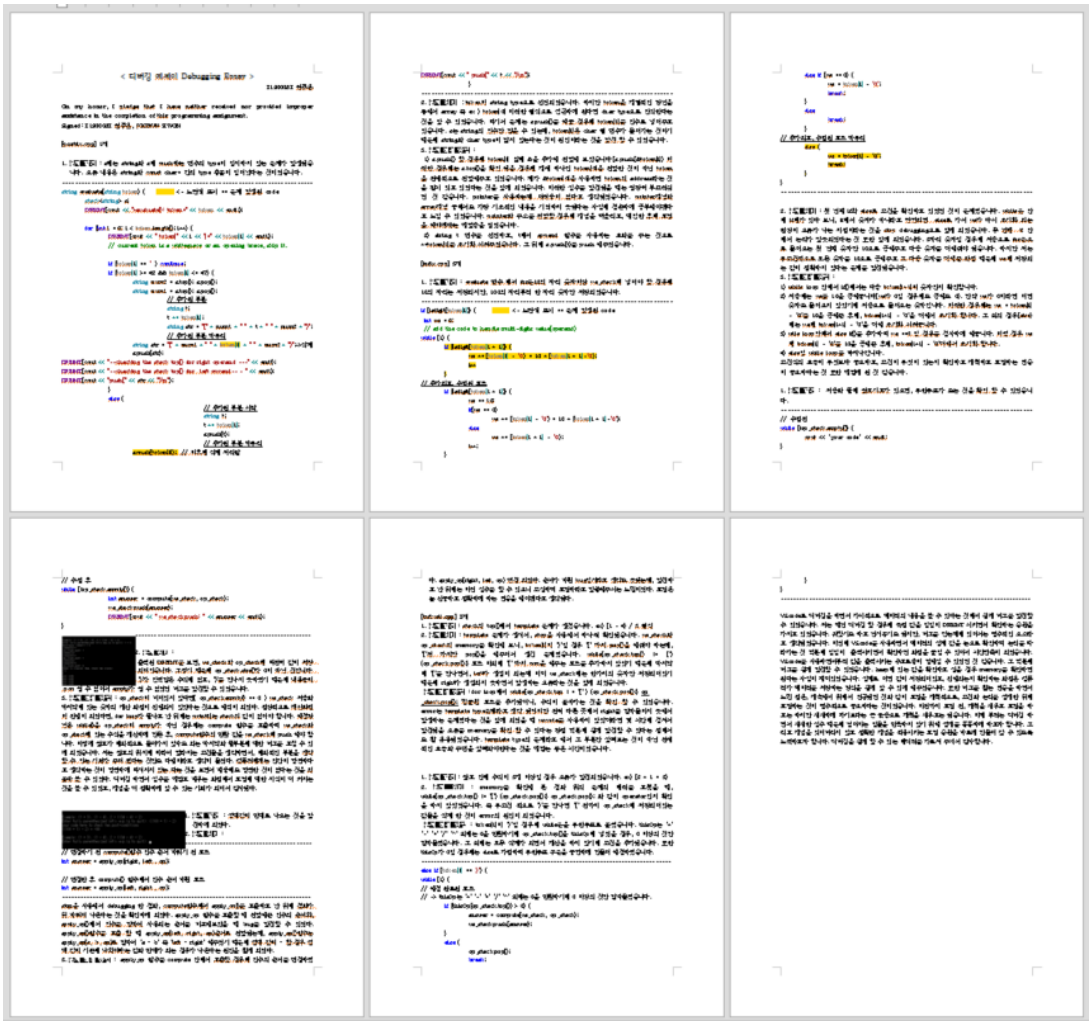
1 읽어올 토큰이 없을 때까지
  1.1 다음 토큰을 가져옵니다.
  1.2 만약 토큰이
    1.2.1 공백일 경우, 무시합니다
    1.2.2 왼쪽 괄호일 경우, operator stack 에 push()합니다.
    1.2.3 숫자일 경우
      1.2.3.1 숫자를 읽어옵니다 (여러 자리 수일 수 있습니다).
      1.2.3.2 operand stack 에 push()합니다.
    1.2.4 오른쪽 괄호일 경우
      1.2.4.1 operator stack 의 맨 위 요소가 왼쪽 괄호가 될 때까지
        1.2.4.1.1 operator stack 에서 연산자를 pop()합니다.
        1.2.4.1.2 operand stack 을 2 번 pop()하여 2 개의 피연산자를 가져옵니다.
        1.2.4.1.3 연산자를 올바른 순서로 피연산자에 적용합니다
        1.2.4.1.4 결과값을 operand stack 에 push()합니다.
      1.2.4.2 operator stack 에서 왼쪽 괄호를 pop()하고 제거합니다.
    1.2.5 연산자(이하 **thisOp**)
      1.2.5.1 operator stack 이 비워지고, operator stack 의 맨 위 항목이 thisOp 보다 더 높거나 같은 **우선순위**를 가질 때까지
        1.2.5.1.1 operator stack 에서 연산자를 pop()합니다.
        1.2.5.1.2 operand stack 을 2 번 pop()하여 2 개의 피연산자를 가져옵니다.
        1.2.5.1.3 연산자를 올바른 순서로 두 값에 적용합니다.
        1.2.5.1.4 결괏값을 operand stack 에 push()합니다.
      1.2.5.2 연산자(thisOp)를 operator stack 에 push()합니다.
2 (전체 식을 모두 분석했으니 op stack 에 남아있는 연산자들을 operand stack 에 남아있는 피연산자들에 적용합니다)
  operator stack 이 비워질 때까지
  2.1 operator stack 에서 연산자를 pop()합니다.
  2.2 operand stack 을 2 번 pop()하여 2 개의 피연산자를 가져옵니다.
  2.3 연산자를 올바른 순서로 두 값에 적용합니다.
  2.4 결괏값을 operand stack 에 push()합니다.
3 (이 시점에서 operator stack 은 비어 있어야 하며, operand stack 에는 단 1 개의 결괏값만 저장되어 있어야 합니다.)
  operand stack 의 맨 위 요소를 반환합니다.

Step 4: MS VS 혹은 Xcode 사용 경험 에세이

VS 또는 XCode 를 사용하면서 느낀 점과 깨달은 점을 (커버 페이지를 제외하고) 적어도 3 페이지 분량의 에세이를 작성하세요.

- 이 에세이의 목적은 자신이 이 tool 을 활용할 줄 안다는 것을 보여주고, 이번에 코딩을 하면서 이 tool 을 어떻게 활용했는지 보여주는 것입니다.
- 스크린 캡처와 코드를 포함할 수 있으며, 이는 **전체 분량의 50% 이상을 차지하지 않아야 합니다.**

- 소수의 좋은 에세이에만 1.5 점을 부여할 것입니다.
  최소한의 요구 사항을 충족한다면 0.5~1.0 점을 부여할 것입니다.
- 에세이의 개요는 다음과 같습니다.



과제 제출

- **On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.**
  서명: _____ 학번: _____
- 제출하기 전에 코드가 제대로 컴파일이 되고 실행되는지 확인하세요. 제출 직전에 급하게 코드를 수정한 후 코드가 제대로 컴파일이 될 거라고 짐작하지 않는 게 좋습니다. “거의” 작동하는 코드도 틀린 것입니다.
- 과제가 컴파일 및 실행된다면, 마감 기한 전까지 과제의 일부만 완성했더라도 제출하기 바랍니다. 마감 시간 이후 24 시간 이내 제출하면, 만점에서 25% 감점하고 채점합니다. 그 이상 늦은 것은 채점하지 않으며, 0 점 처리합니다.
- 제출 후, **마감 기한 전까지** 수정 및 재제출이 가능합니다. 파일 하나만 수정하더라도 해당 파일과 관련된 파일들을 모두 재제출해야 합니다. 재제출 횟수는 제한 없습니다. 마감 기한 전에 **가장 마지막으로** 제출된 파일을 채점할 것입니다.

제출 파일 목록

드라이버 파일은 제출하지 마세요. 다음 파일들은 각자의 드라이버 파일을 수정하지 않아도 잘 작동해야 합니다.

- postfix.cpp
- infix.cpp
- infixall.cpp

마감 기한 & 배점

- 마감일: 11:55 pm
- 점수:
  - 1 단계: 1 점
  - 2 단계: 1 점
  - 3 단계: 1 점
  - 4 단계: 0.5 ~ 1.0 점
    (소수의 좋은 에세이는 1.5 점까지 받을 수 있습니다)