

# MyCalculator

## 프로젝트

과목: C언어

담당 교수: 황성운

11조

조장: 김나연(스마트보안전공, 202435143)

조원:

박이진(스마트시티융합학과, 202335230)

오시현(스마트보안전공, 202435166) 김

현아(스마트보안전공, 202435158)

작성자: 김나연

제출일: 2024년 11월 24일

## 목차

### I. 서론

1. 프로젝트 개요
2. 프로젝트 진행 방식
3. 프로젝트 일정

### II. 본론

1. 이항 연산 분석 및 디버깅
2. 초기 다항 연산 분석 및 문제 해결 과정
3. 최종 다항 연산 구조 분석
4. 최종 다항 연산 디버깅 및 결과 분석

### III. 결론

1. 느낀 점 및 학습 내용
2. 향후 개선 방안

### IV. 부록

1. 회의 기록
2. 구성원의 역할 분담 및 기여점

## I. 서론

### 1. 프로젝트 개요

본 프로젝트는 문자열 계산기 프로그램을 제작함으로써 협업 능력 및 리더십을 기르는 것을 목적으로 한다. 프로젝트의 주요 기능은 다음과 같다.

- 간단한 사칙 연산을 제공: 덧셈, 뺄셈, 곱셈, 나눗셈의 기본 연산 구현.
- 이항 및 다항 연산 지원: 두 개 이상의 문자열을 처리하는 연산 구현.

### 2. 프로젝트 진행 방식

#### ■ 진행 순서

#### 1) 개별 이항 연산 구현

- 각 조원이 사칙 연산(덧셈, 뺄셈, 곱셈, 나눗셈)을 선택하여 이항 연산을 구현.

#### 2) 코드 취합 및 다항 연산 아이디어 제시

- 구현된 네 개의 코드를 취합한 뒤, 다항 연산 구현을 위한 각자의 아이디어를 공유.

#### 3) 아이디어 반영 및 코드 제작

- 취합한 코드를 바탕으로 각자의 아이디어에 맞는 다항 연산 기능을 추가하여 코드를 제작.

#### 4) 코드 공유 및 최종본 투표

- 모든 조원이 제작한 네 개의 코드를 공유하고, 프로젝트 목적에 가장 적합한 최종 코드를 투표로 선정.

#### 5) 분할 컴파일 수행

- 가장 많은 투표를 받은 코드의 소스 파일을 분할 컴파일함으로써 효율적인 환경 구성.

### 3. 프로젝트 일정

#### 1) 2024년 11월 5일 ~ 2024년 11월 8일

- 개별 이항 연산 구현

#### 2) 2024년 11월 9일 (1차 회의)

- 구조체 사용 여부, 다항 연산 구현 방법 및 문장 입출력 방식 논의

#### 3) 2024년 11월 9일 ~ 2024년 11월 14일

- 개별 다항 연산 구현

#### 4) 2024년 11월 15일 (2차 회의)

- 괄호 사용 규칙 논의 및 개별 코드 설명

#### 5) 2024년 11월 15일 ~ 2024년 11월 19일

- 개별 다항 연산 구현 개선

#### 6) 2024년 11월 19일 (3차 회의)

- 개별 다항 연산 재구현 논의 및 아이디어 재정비

#### 7) 2024년 11월 19일 ~ 2024년 11월 22일

- 개별 다항 연산 구현 재구현

#### 8) 2024년 11월 22일 (4차 회의)

- 투표로 최종본 선정 및 분할 컴파일 수행

## II. 본론

### 1. 이항 연산 분석 및 디버깅

본 분석은 문자열 계산을 토대로 한다. 문자열 연산을 보다 명확히 설명하기 위해, 본 보고서에서는 문자열 연산에서 사용하는 각 문자열에 대해 별도의 용어를 정의한다.

1. 덧셈 연산: 덧셈 연산에서는 각각의 문자열을 '피가문자열'과 '가문자열'로 지칭하도록 한다. 이는 숫자 연산에서 '피가수', '가수'를 사용하는 것과 유사한 개념이다.
2. 그 외 연산: 덧셈 이외의 연산에서도 유사한 용어(피감문자열, 감문자열 등)를 사용한다.

### ■ 이항 연산 분석

#### 1) main, find\_op

```
int main() {
    //기본 문자열 입력받기 => (문자열) (연산기호) (문자열) 형식
    printf("문자열을 입력하세요: ");
    gets_s(sen, sizeof(sen));

    //연산 기호 찾기 1 (참조에 의한 호출)
    find_op(sen);

    //기호를 기준으로 왼쪽의 문자열 추출
    token = strtok_s(sen, "+*/()", &next_sen);

    //문자열 추출이 됐다면 아래 코드 실행하기
    while (token != NULL && op != 'W0' && *next_sen != 'W0') {

        //각 토큰 앞 쪽 공백 제거하기
        while (*token == ' ') token++;
        while (*next_sen == ' ') next_sen++;

        //계산하기
        calculate(sen, token, next_sen, op, result, count, word);

        token = strtok_s(NULL, "+*/()", &next_sen);
    }

    return 0;
}

//연산 기호 원지 알기
void find_op(char* sentence) {
    char op_list[] = "+*/";
    int i = 0;

    //맨 처음에
    do {
        char* pos = strchr(sentence, op_list[i]);
        if (pos != NULL) {
            op = op_list[i];
            break;
        }
        i++;
    } while (i < sizeof(op_list) - 1);
}
```

- 문자열을 입력받아 **sen**에 저장한다.
- **sen**에서 연산기호를 찾는다.
  - (1) do-while문을 사용하여 문자열(sen)에 연산기호가 포함되어 있는지 확인한다.
  - (2) op\_list에 나열된 연산기호 중 가장 왼쪽에 있는 연산기호를 검색한다.
  - (3) 연산기호를 발견하면 **op**에 저장하고 탐색을 종료한다.
- 연산기호를 기준으로 분리한 앞부분 문자열을 문자열 포인터 **\*token**에 저장한다.
- 토큰의 앞 공백을 제거하고 계산을 수행하는 것을 분할 문자열이 없을 때까지 반복한다.

## 2) 덧셈

```
//덧셈 함수
void my_add(char* a, char* b, char* result) {
    result[0] = '\0';
    strcat(result, a);
    strcat(result, b);
}
```

- 피가문자열(a)와 가문자열(b), 결과 문자열(result)을 매개변수로 받는다.
- 결과 문자열을 초기화하여 이후 연산이 정상적으로 수행되도록 한다.
- 피가수와 가수를 차례로 결과 문자열의 끝에 붙여 넣는다.

## 3) 뺄셈

```
//뺄셈 함수
void my_sub(char* a, char* b, char* result) {
    result[0] = '\0'; //결과 문자열 초기화
    char* pos = strstr(a, b);
    if (pos != NULL) {
        int len = strlen(b);
        strncpy(result, a, pos - a);
        result[pos - a] = '\0';
        strcat(result, pos + len);
    }
    else {
        strcpy(result, a);
    }
}
```

- 피감문자열(a)와 감문자열(b), 결과 문자열(result)을 매개변수로 받는다.
- 결과 문자열을 초기화하여 이후 연산이 정상적으로 수행되도록 한다.
- **strstr** 함수를 사용하여 피감문자열에서 감문자열을 찾고, 감문자열의 시작 주소를 문자열 포인터 **pos**에 저장한다.
- 감문자열을 찾은 경우:
  - (1) 감문자열의 시작 부분까지를 결과 문자열에 복사한다.
  - (2) 감문자열이 끝난 이후의 나머지 부분을 결과 문자열에 붙여 넣는다.
- 감문자열을 찾지 못한 경우:
  - (1) 피감문자열 전체를 결과 문자열에 복사한다.

#### 4) 곱셈

```
//곱셈 함수
void my_multiply(char* a, int b, char* result) {
    result[0] = '\0'; //결과 문자열 초기화
    for (int i = 0; i < b; i++) {
        strcat(result, a);
    }
}
```

- 피승문자열(a)와 승수(b), 결과 문자열(result)을 매개변수로 받는다.
- 결과 문자열을 초기화하여 이후 연산이 정상적으로 수행되도록 한다.
- 승수값만큼 반복문이 실행되며 피승문자열을 결과 문자열 끝에 붙여 넣는다.

#### 5) 나눗셈

```
//나눗셈 함수
void my_divide(char* a, char* b, int* count) {
    int result = 0; //결과값 초기화

    char* pos = strstr(a, b);
    while (pos != NULL) {
        result++;
        pos = strstr(pos + strlen(b), b); //현재 위치 이후로 계속 검색
    }

    *count = result;
}
```

- 피제문자열(a)와 피제문자열(b), 결과값(count)을 매개변수로 받는다.
- 정수형 결과값을 초기화하여 이후 연산이 정상적으로 수행되도록 한다.
- strstr 함수를 사용하여 피제문자열에서 제문자열을 찾고, 제문자열의 시작 주소를 문자열 포인터 pos에 저장한다.
- 제문자열을 찾은 경우:
  - (1) 정수형 결과값을 1씩 증가시킨다.
  - (2) 검색 위치를 제문자열 이후로 이동시킨다.
  - (3) 반복문이 종료되면 제문자열이 피제문자열에 포함된 총 횟수를 결과값(\*count)에 저장한다.
- 감문자열을 찾지 못한 경우:
  - (1) 정수형 결과값(0)을 그대로 결과값(\*count)에 저장한다.

## ■ 이항 연산 디버깅

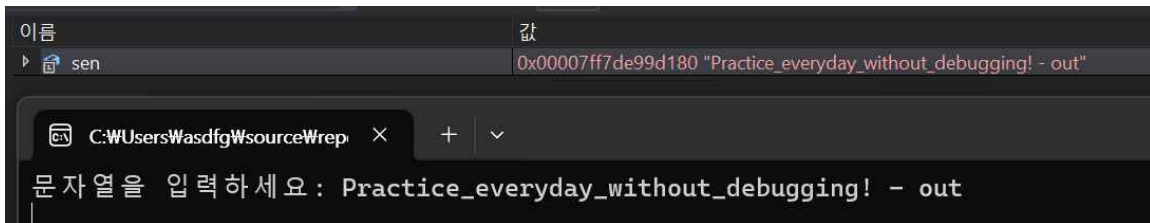
1) main, find\_op

<main>

문자열을 입력하세요 : |

```
//기본 문자열 입력받기 => (문자열) (연산기호) (문자열) 형식
printf("문자열을 입력하세요: ");
gets_s(sen, sizeof(sen));
```

- 문자열을 입력받아 sen에 저장한다.



```
//연산 기호 찾기 1 (참조에 의한 호출)
find_op(sen); 경과 시간 723,247ms 이하
```

<find\_op>

```
24
25 //연산 기호 뭉치 알기
26 void find_op(char* sentence) { 경과 시간 1ms 이하
27
28     char op_list[] = "+-*/";
29     int i = 0;
30
31     //맨 처음에
32     do {
33         char* pos = strchr(sentence, op_list[i]);
34         if (pos != NULL) {
35             op = op_list[i];
36             break;
37         }
38         i++;
39     } while (i < sizeof(op_list) - 1);
40 }
41
```

- sen(문자열)에서 연산기호를 찾는다.

```
char op_list[] = "+-*/";
int i = 0;
```

- op\_list(연산기호 문자 배열)와 i(배열의 인덱스)를 선언 및 초기화한다.

```
//맨 처음에
do {
    char* pos = strchr(sentence, op_list[i]);
    if (pos != NULL) { 경과 시간 1ms 이하
        op = op_list[i];
        break;
    }
}
```

- op\_list의 i번째 문자를 sentence(문자열)에서 찾는다.
  - op\_list[i]가 '+'일 때, pos는 NULL이다(sentence에서 '+'를 찾지 못했다).

op_list[i]	43 '+'
pos	0x0000000000000000 <NULL>
sentence	0x00007ff7de99d180 "Practice_everyday_without_debugging! - out"

pos가 NULL이므로 if문을 건너뛰고 i를 1 증가시킨다.

(op\_list의 크기 - 1)(=4)보다 i(=1)가 작기 때문에, while문을 타고 다시 do로 올라간다.

```
printf("%d", sizeof(op_list));
```

op_list	0x000000b6300ff4d4 "+-*/"	char[5]
[0]	43 '+'	char
[1]	45 '-'	char
[2]	42 '*'	char
[3]	47 '/'	char
[4]	0 '\0'	char

- ♦ 이때 op\_list의 크기는 5이다. 배열의 값이 '\0'을 포함하여 5개이기 때문이다.



```
char* pos = strchr(sentence, op_list[i]);
```

▶ op_list	0x0000008cf0effb84 "+-*/"	Q ▼	char[5]
op_list[i]	45 '-'		char
▶ pos	0x00007ff7f95cd1a5 "- out"	Q ▼	char *
	45 '-'		char
▶ sentence	0x00007ff7f95cd180 "Practice_everyday_without_debugging! - out"	Q ▼	char *

- op\_list[i]가 '-'일 때, pos는 '-'를 가리킨다(sentence에서 '-'를 찾았다).

```
if (pos != NULL) {
    op = op_list[i];
    break; 경과 시간 1ms 이하
```

- pos가 NULL이 아니므로 if문을 수행하고 while문을 중지한다.

op	45 '-'		char
▶ op_list	0x0000008cf0effb84 "+-*/"	Q ▼	char[5]
op_list[i]	45 '-'		char

- op에 op\_list[i]가 저장된다.

```
//기호를 기준으로 왼쪽의 문자열 추출
token = strtok_s(sen, "+-*/( )", &next_sen);
```

이름	값	형식
*next_sen	32 ''	char
op	45 '-'	char
▶ token	0x00007ff7f95cd180 "Practice_everyday_without_debugging!"	Q ▼ char *

- strtok\_s에 의해 token에 "Practice\_everyday\_without\_debugging!"이 저장되었다.

<main>

```
//문자열 추출이 됐다면 아래 코드 실행하기
while (token != NULL && op != 'WO' && *next_sen != 'WO') {

    //각 토큰 앞 쪽 공백 제거하기
    while (*token == ' ') token++; 경과 시간 1ms 이하
    while (*next_sen == ' ') next_sen++;
```

- token(피연산문자열), op(연산기호), next\_sen(연산문자열)이 모두 존재할 때, 각 문자열의 앞 쪽 공백을 제거한다.
- 이렇게 공백을 제거함으로써 안전한 연산이 수행되게 한다.

<공백 제거 이전>

이름	값	형식
*next_sen	32 ''	char
*token	80 'P'	char
next_sen	0x00007ff7f95cd1a6 " out"	char *
token	0x00007ff7f95cd180 "Practice_everyday_without_debugging! "	char *

<공백 제거 이후>

이름	값	형식
token	0x00007ff7f95cd180 "Practice_everyday_without_debugging! "	char *
*token	80 'P'	char
next_sen	0x00007ff7f95cd1a7 "out"	char *
*next_sen	111 'o'	char

op
45 '-'

```
//뿔셈
else if (op == '-') {
    sub(token, next_sen, result);
    printf("%s\n", result);
}
```

```
//뿔셈 함수
void sub(char* a, char* b, char* result) {
    result[0] = 'WO'; //결과 문자열 초기화
    char* pos = strstr(a, b);
    if (pos != NULL) {
        int len = strlen(b);
        strncpy(result, a, pos - a);
        result[pos - a] = 'WO';
        strcat(result, pos + len);
    }
    else {
        strcpy(result, a);
    }
}
```

- op가 '-'이므로 뿔셈을 수행한다.
- 연산 방법은 다항 연산 구현 코드와 유사하니 생략하였다.

next_sen	0x00000058201ef8d7 "out"	char *
op	45 '-'	char
result	0x00000058201efa00 "Practice_everyday_with_debugging! "	char[300]
token	0x00000058201ef8b0 "Practice_everyday_without_debugging! "	char *

- result가 "Practice\_everyday\_with\_debugging! "가 된 것을 볼 수 있다.

<최종 결과>

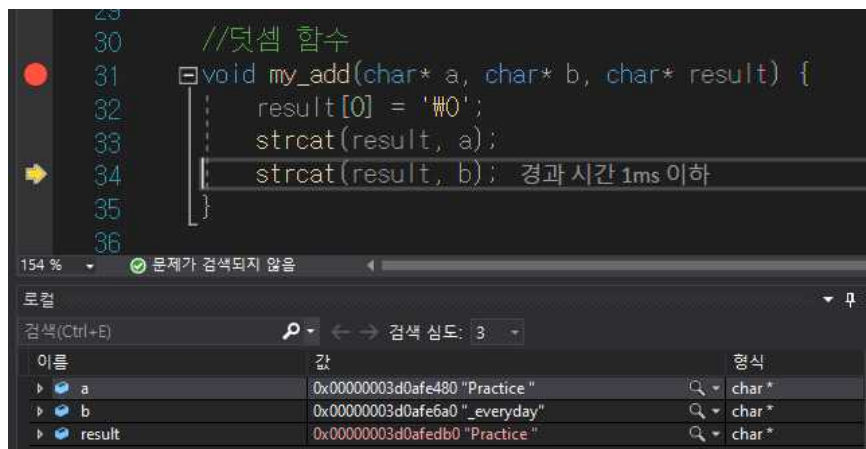
문자열을 입력하세요 : Practice\_everyday\_without\_debugging! - out  
Practice\_everyday\_with\_debugging!

## 2) 덧셈

문자열을 입력하세요 : Practice+\_everyday

이름	값	형식
a	0x00000003d0afe480 "Practice "	char *
b	0x00000003d0afe6a0 "_everyday"	char *
result	0x00000003d0afedb0 ""	char *

- 피가문자열(a)에 "Practice ", 가문자열(b)에 "\_everyday"가 저장되었다.
- 결과 문자열은 빈 문자열로 초기화되었다.



- 결과 문자열에 피가수가 붙여 넣어졌다.

```

30 //덧셈 함수
31 void my_add(char* a, char* b, char* result) {
32     result[0] = '\0';
33     strcat(result, a);
34     strcat(result, b);
35     } 경과 시간 1ms 이하
36

```

154 % 문제가 검색되지 않음

로컬

검색(Ctrl+E) 검색 심도: 3

이름	값	형식
a	0x0000003d0afe480 "Practice "	char *
b	0x0000003d0afe6a0 "_everyday"	char *
result	0x0000003d0afedb0 "Practice_everyday"	char *

- 결과 문자열에 가수가 붙여 넣어졌다.

<최종 결과>

문자열을 입력하세요: Practice+\_everyday  
최종 결과: Practice\_everyday

### 3) 뺄셈

문자열을 입력하세요: Practice\_everyday\_without\_debugging! - out

이름	값	형식
a	0x0000001a812fe370 "Practice_everyday_without_debugging! "	char *
b	0x0000001a812fe590 "out"	char *
result	0x0000001a812feca0 ""	char *
result[0]	0 '\0'	char

- 피감문자열(a)에 “Practice\_everyday\_without\_debugging! ”, 가문자열(b)에 “out”이 저장되었다.
- 결과 문자열은 빈 문자열로 초기화되었다.

이름	값	형식
a	0x0000001a812fe370 "Practice_everyday_without_debugging! "	char *
b	0x0000001a812fe590 "out"	char *
pos	0x0000001a812fe386 "out_debugging! "	char *
result	0x0000001a812feca0 ""	char *
result[0]	0 '\0'	char

- strstr 함수를 사용하여 피감문자열에서 감문자열을 찾는다.

```

39
40 //뺄셈 함수
41 void my_sub(char* a, char* b, char* result) {
42     result[0] = '\0'; //결과 문자열 초기화
43     char* pos = strstr(a, b);
44     if (pos != NULL) { 경과 시간 1ms 이하
45         int len = strlen(b);
46         strncpy(result, a, pos - a);
47         result[pos - a] = '\0';
48         strcat(result, pos + len);
49     }
50     else {
51         strcpy(result, a);
52     }
53 }

```

pos	0x0000001a812fe386 "out_debugging! "
111 'o'	

- pos에 감문자열의 시작 문자인 'o'가 저장되었다.

```

45 int len = strlen(b);
46 strncpy(result, a, pos - a); 경과 시간 1ms 이하

```

154 % 문제해결 검색되지 않음

조사식 1

검색(Ctrl+E) 검색 심도: 3

이름	값	형식
len	3	int

- len에 감문자열의 길이를 저장한다. 이때 "out"의 길이인 3이 저장된다.

```

strncpy(result, a, pos - a);

```

이름	값	형식
a	0x0000001a812fe370 "Practice_everyday_without_debugging! "	char *
b	0x0000001a812fe590 "out"	char *
len	3	int
pos	0x0000001a812fe386 "out_debugging! "	char *
result	0x0000001a812feca0 "Practice_everyday_with"	char *

- result(결과값)에 감문자열 앞의 문자열을 복사한다.

```
strcat(result, pos + len);
```

이름	값	형식
▶ a	0x00000049d2bbe420 "Practice_everyday_without_debugging! "	char *
▶ b	0x00000049d2bbe640 "out"	char *
▶ pos	0x00000049d2bbe436 "out_debugging! "	char *
▶ result	0x00000049d2bbbed50 "Practice_everyday_with_debugging! "	char *

- result(결괏값)에 감문자열 끝의 문자열을 붙여 넣는다.

이름	값	형식
▶ a	0x000000e38ddfe9d0 "Practice_everyday_with_debugging! "	char *
▶ b	0x000000e38ddfeb0 "out"	char *
▶ pos	0x0000000000000000 <NULL>	char *

else {	이름	값
strcpy(result, a); 경과 시간 1ms 이하	▶ a	0x000000e38ddfe9d0 "Practice_everyday_with_debugging! "
}	▶ result	0x000000e38ddff300 "Practice_everyday_with_debugging! "

- pos가 NULL이라면(피감문자열에서 감문자열을 찾지 못했다면), 피감문자열을 result(결괏값)에 복사한다.

<최종 결과>

```
문자열을 입력하세요: Practice_everyday_without_debugging! - out
최종 결과: Practice_everyday_with_debugging!
```



#### 4) 곱셈

문자열을 입력하세요: \_Practice! \* 3

이름	값	형식
a	0x000000df186fe530 "_Practice! "	char *
b	3	int
result	0x000000df186fee60 ""	char *

- 피승문자열(a)에 "\_Practice! ", 승수(b)에 3이 저장되었다.
- 결과 문자열은 빈 문자열로 초기화되었다.

```

58 //곱셈 함수
59 void my_multiply(char* a, int b, char* result) {
60
61     result[0] = '\0'; //결과 문자열 초기화
62     for (int i = 0; i < b; i++) { 경과시간 1ms 이하
63         strcat(result, a);
64     }
65 }

```

이름	값	형식
*time	3	int
pre_sen	0x000000df186fe530 "_Practice! "	char *
result	0x000000df186fee60 "_Practice! _Practice! _Practice! "	char *

- 승수(b)만큼 반복하며 result(결과값)에 피승문자열을 붙여 넣는다.

#### <최종 결과>

문자열을 입력하세요: \_Practice! \* 3  
최종 결과: \_Practice! \_Practice! \_Practice!

## 5-1) 기존의 나눗셈 구현과 문제

```
//나눗셈
else if (op == '/') {
    word = strtok_s(NULL, "+*/()", &next_sen); // 다음 토큰(개수를 셀 단어) 반환

    divide(token, word, &count); //참조에 의한 호출
    printf("%d\n", count);
}
```

```
//나눗셈 함수
void divide(char* a, char* b, int* count) {
    int result = 0; //결과값 초기화

    //최대 100개의 단어, 단어 길이는 100자까지.
    char save[100][100];

    char seps[] = " ,.tWn"; //분리자
    char* pch = strtok(a, seps); //첫 단어 분리
    int i = 0; //배열의 i번째 행

    //단어 분리 후 2차원 배열에 저장
    while (pch != NULL) {
        strcpy_s(save[i], sizeof(save[i]), pch);
        i++;
        pch = strtok(NULL, seps); // 현재 위치 이후로 계속 검색
    }

    //분리된 단어중 word와 같은 단어 개수 세기
    for (int j = 0; j < i; j++) {
        if (strcmp(save[j], b) == 0) {
            result++;
        }
    }

    *count = result;
}
```

- 기존의 나눗셈 구현 방식은 `strtok`를 사용해 연산기호와 괄호를 기준으로 문자열을 분리한 뒤, `word`(제문자열)과 `pch`(문자열의 각 단어들)을 `strcmp`를 사용해 비교하며 같을 시에 `count`(결과값)을 1씩 증가하는 방식이었다.
- 하지만 해당 방법은 단어를 분리하여 비교하는 방식으로, 예제에 맞게 수행되지 않는다는 문제가 발생하였다.

▷ save[j]	0x000000953acfc504 "programming"	Q ▾	char[100]
▷ b	0x000000953acff2ee "program"	Q ▾	char *
▷ result	0		int
▷ save[j]	0x000000953acfa18 "programmer"	Q ▾	char[100]
▷ b	0x000000953acff2ee "program"	Q ▾	char *
▷ result	0		int

- “programming”과 “programmer”는 “program”과 같지 않아 결과값에 포함되지 않고, 0이 출력된다.

해당 문제를 해결하기 위해 교수님께 조언을 구했고, 그를 바탕으로 다음과 같은 방식으로 나눗셈 연산을 재구현하였다.



## 5-2) 개선된 나눗셈

문자열을 입력하세요: Practice programming at least 30 minutes every day, and you will be an advanced programmer. / program

이름	값	형식
a	0x000000a161b9e240 "Practice programming at least 30 minutes every day, and you will be an advanced programmer. "	char *
b	0x000000a161b9e460 "program"	char *

- a(피제문자열)에 “Practice programming at least 30 minutes every day, and you will be an advanced programmer. ”, b(제문자열)에 “program”이 저장되었다.
- 결과 문자열은 빈 문자열로 초기화되었다.

```

70 //나눗셈 함수
71 void my_divide(char* a, char* b, int* count) {
72     int result = 0; //결과값 초기화
73
74     char* pos = strstr(a, b);
75     while (pos != NULL) { 경과 시간 1ms 이하
76         result++;
77         pos = strstr(pos + strlen(b), b); //현재 위치 이후로 계속 검색
78     }
79
80     *count = result;
81 }

```

pos	0x000000a161b9e249 "programming at least 30 minutes every day, and you will be an advanced programmer. "	char *
112 p		char

- strstr 함수를 사용하여 피제문자열에서 제문자열을 찾고, pos에 저장한다.
- pos가 NULL이 될 때까지 while문을 반복한다.

pos	0x000000a161b9e249 "programming at least 30 minutes every day, and you will be an advanced programmer. "
result	1

- 만약 제문자열을 찾았다면 result를 1씩 증가한다.

```
pos = strstr(pos + strlen(b), b); //현재 위치 이후로 계속 검색
```

b	0x000000a161b9e460 "program"
pos	0x000000a161b9e290 "programmer. "

- 이전에 찾은 제문자열 이후부터 다시 제문자열을 찾는다.
- 해당 과정을 pos가 NULL이 될 때까지 반복한다.

```
*count = result;
```

*count	2	int
result	2	int

- while문을 종료한 뒤 매개변수인 \*count(결괏값)에 result(정수형 결괏값)을 저장한다.

<최종 결과>

```
문자열을 입력하세요: Practice programming at least 30 minutes every day,
and you will be an advanced programmer. / program
최종 결과: 2
```

## 2. 초기 다항 연산 분석 및 문제 해결 과정

### 1) 김나연(스마트보안전공 202435143) (최종본)

최종본이기에 자세한 코드 설명은 생략한다. 이후에 코드 분석과 디버깅이 이루어진다.

- 배열을 문자열과 비교하는 과정에서 값이 제대로 출력하지 않는 문제가 발생하였다.

<문제 사진>

```
// 결과 출력
if (my_op == "+" || "-" || "*") printf("섹션 %d 결과: %s\n", i + 1, result);
else if (my_op == "/") printf("섹션 %d 결과: %d\n", i + 1, count);
문자열을 입력하세요: i love you and you love me / love
최종 결과:
```

- 문자열을 비교할 때는 <string.h> 라이브러리의 strcmp를 사용해야 하는 것을 간과하여 생긴 문제였다.
- 고로 아래와 같이 단일 문자 op를 단일 문자와 비교하는 조건식으로 수정했다.

<해결 사진>

```
//최종 결과 출력
void final_output(char* result, int count, char op) {
    if (op == '/') printf("최종 결과: %d\n", count);
    else printf("최종 결과: %s\n", result);
}
```

## 2) 박이진(스마트시티융합학과, 202335230)

```
// 괄호 처리 함수
void function(char* sen, char* result) {
    char temp[700];
    int i = 0, j = 0, k = 0;
    char op = '#0';
    char a[700] = "";
    char b[700] = "";
    char before[500], after[500];

    if (strchr(sen, '(') != NULL) {
        for (i = 0; sen[i] != '\0'; i++) {
            if (sen[i] == '(') j = i; //현재문자가 (일 경우 j에 저장
            else if (sen[i] == ')') {
                int length = i - j - 1;
                strncpy(temp, sen + j + 1, length); //sen+j+1: ( 바로 다음 문자
                temp[length] = '\0'; //i-j-1: 괄호 안에 있는 문자열 길이

                function(temp, result);

                strncpy(before, sen, j); //before: (괄호 이전 부분
                before[j] = '\0';

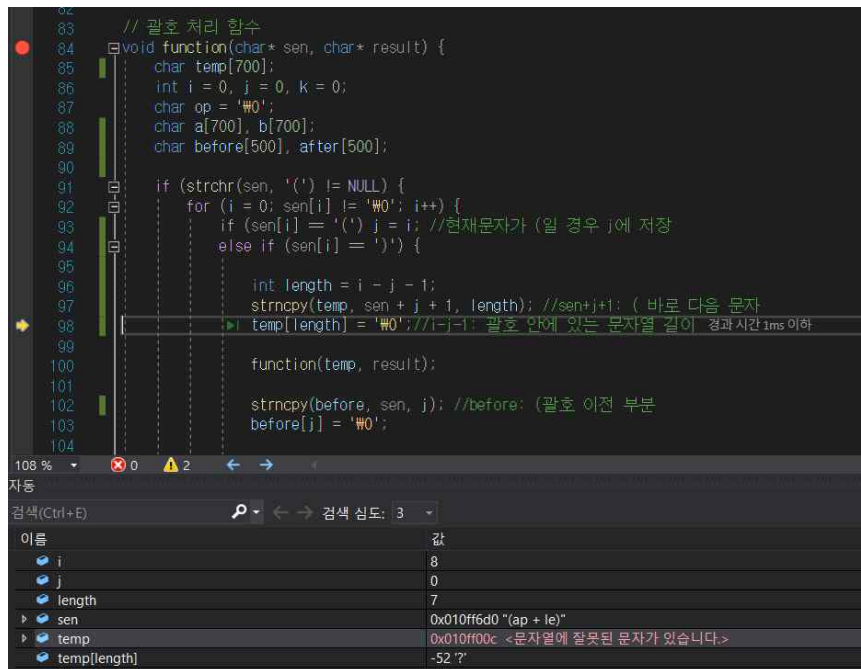
                strcpy(after, sen + i + 1); // after: ) 괄호 이후 부분

                strcpy(temp, before);
                strcat(temp, result);
                strcat(temp, after);

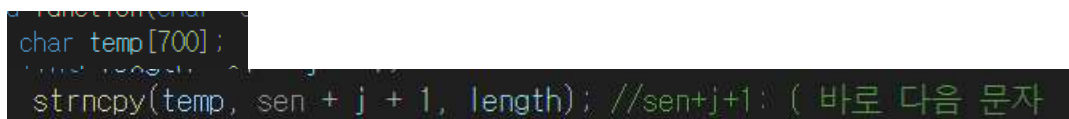
                strcpy(sen, temp);
                i = -1;
            }
        }
    }
}
```

- 문자열에서 열린 괄호와 닫힌 괄호 사이의 문자열을 추출하여 계산한다.
- 괄호 안에 괄호가 존재하는 경우, 반복 루프와 문자열 분리를 통해 다시 계산한다.
- 이후 괄호가 더 없다면 문자열 내의 연산자를 찾아서 계산한다.

<문제>



- temp가 제대로 동작하지 않는 오류가 발생하였다.
- 해당 문제는 배열을 초기화 하지 않은 상태에서 함수 동작에 활용함으로써 생긴 오류였다.



이름	값	형식
i	4	int
result	0x000000ae54eff320 "_Practice ㄷㄸㄹㄺㄻㄼㄽㄾㄿㅀㅁㅂㅃㅅㅆㅈㅉㅊㅋㅌㅍㅎ...Q	char *
result[0]	95 '_'	char
str	0x000000ae54ecc30 "_Practice ㄷㄸㄹㄺㄻㄼㄽㄾㄿㅀㅁㅂㅃㅅㅆㅈㅉㅊㅋㅌㅍㅎ... Q	const char *
strcat	0x00007ff7598c655e {igean.exe!strcat}	void *
times	31	int

- 여기서 **temp**는 괄호를 처리하는 과정에서 문자열을 재구성한 배열이다.
  - 이때 배열을 선언하고 초기화를 하지 않고 `strncpy`, `strcat` 등의 함수를 사용하였기 때문에 `_Practice` 뒤에 쓰레기 값이 들어가 있는 것을 알 수 있었다.
  - 또한 이와 같이 다른 문자열 함수들도 모두 같은 오류를 발생하고 있음을 알 수 있었다.
- 
- 해당 이유를 토대로, 배열을 선언하면 무조건 초기화 해주어 안전한 계산이 이루어지도록 코드를 수정하였다.

```
// 괄호 처리 및 연산자 처리 함수
void function(char* sen, char* result) {
    char temp[700] = "";
    char before[700], after[700];
    char a[700] = "", b[700] = "", middle[700] = "";
    int i = 0, j = 0, k = 0;
```

### <문제 해결 과정>

- 팀장과의 피드백을 통해 문제를 해결하였다.

1:44 지금 제가 보기엔 배열들을 초기화 시키지 않은 상태에서 안에 값을 복사해서 그런 것 같거든요

1:45 기본적으로 전역변수가 아닌 이상 지역변수들은 자동초기화가 안되었고 안에 쓰레기값이 들어가있어요

오후 1:46

근데 여기서 function 함수 내부의 배열을 초기화 하지 않은 상태에서 \_Practice를 복사하니까 당연히 쓰레기값 위에 문자열이 없어지겠죠?

그래서 9개의 문자와 691개의 쓰레기값이 들어가있는 배열을 result에 31번이나 붙이라고 하면 당연히 버퍼 오버플로우가 발생하겠죠

그래서 저는 보통 c에서는 웬만한 지역 변수, 포인터, 배열 선언할 때는 다 초기화 한 상태로 해요

나중에 수정하기 좋게 배열은 a[SIZE] 로 해놓고

오후 1:48 가장 위에 #define SIZE 500 이런 식으로 해놓고요

오후 1:49

결론은 이렇게 배열만 초기화 해줘도 바로 해결됩니다

근데 보니까 공백은 잘 들어가는 것 같은데 나머지 연산은 잘 안 되고 있는 것 같네요... 출력도 잘 안 되구요

## 3) 오시현(스마트보안전공, 202435166)

```
// 괄호 내외 내용 분리
void bracket(char* sen, char in[20][500]) {
    char temp[500];
    int idx = 1;
    char copy[500];
    strcpy(copy, sen);
    int open = 0, close = 0;

    while (strchr(copy, '(') != NULL) {
        bracket_(copy, temp, &open, &close);
        if (strlen(temp) > 0) {
            strcpy(in[idx++], temp);
            for (int i = open; i <= close; i++) {
                copy[i] = ' ';
            }
        }
    }

    if (close != -1) {
        strcpy(in[0], copy + close + 1);
    }
}
```

입력된 수식을 괄호 단위로 분리한 후 각각의 연산을 수행한다.

- 괄호가 2개 이상인 다항 연산을 처리하기 위해서 ')'를 찾는다
- 그에 대응하는 '('를 찾아서 괄호 안의 내용을 배열에 저장한다.
- 중간 연산 결과를 계속해서 갱신한다.

```
// 다항 연산 처리 함수
char* cal(char* sen, char in[20][500]) {
    static char pro[500];
    char* result_in = calculator(in[1]);
    strncpy(pro, result_in, sizeof(pro) - 1);
    pro[sizeof(pro) - 1] = '\0';

    // 중첩 괄호 계산
    for (int i = 2; i < 20 && strlen(in[i]) > 0; i++) {
        strncat(pro, in[i], sizeof(pro) - strlen(pro) - 1);
        char* process_result = calculator(pro);
        strncpy(pro, process_result, sizeof(pro) - 1);
        pro[sizeof(pro) - 1] = '\0';
    }

    //외부 내용 결합
    strncat(pro, in[0], sizeof(pro) - strlen(pro) - 1);
    return calculator(pro);
}
```

- 괄호와 외부 내용을 조합하여 다항식을 처리한다.

#### <문제>

- 연산을 실행하는 **calculator** 함수를 별도로 제작하니, 한 번 **result**(결괏값)를 출력하면 함수가 종료되며 지역 문자열 배열인 **result**가 삭제되어 다항 연산이 제대로 동작하지 않는 문제가 발생하였다.

#### <문제 해결 과정>

```
// 기초 연산 과정 처리 함수
char* calculator(char* sen) {
    static char result[500];
```

- result**를 정적 배열로 선언하여 **calculator** 함수가 종료되더라도 다항 연산을 위한 결괏값이 계속 배열 안에 남아있도록 하였다.
- 이로써 다항 연산의 동작이 원활하게 이루어질 수 있었다.



4) 김현아(스마트보안전공, 202435158)

```
// 괄호 처리 함수
void process(char* expression, char* result) {
    char temp[1000] = "";

    while (1) {
        char* open_pos = strrchr(expression, '(');
        char* close_pos = NULL;

        if (open_pos != NULL) {
            close_pos = strchr(open_pos, ')');
        }

        if (open_pos == NULL || close_pos == NULL) break;

        char inner[1000] = "";
        strncpy(inner, open_pos + 1, close_pos - open_pos - 1);
        inner[close_pos - open_pos - 1] = '\0';

        char inner_result[1000] = "";
        process_operation(inner, inner_result);

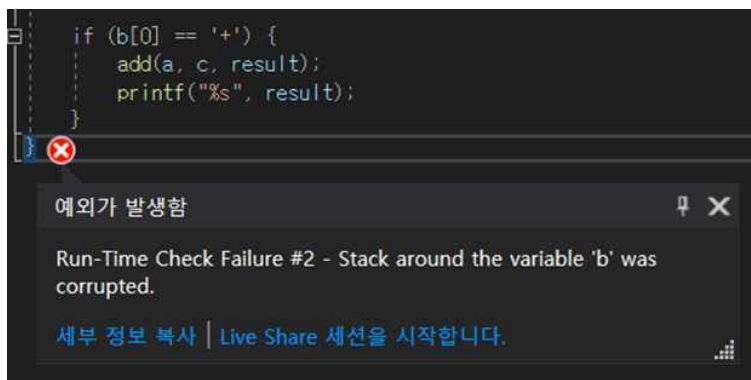
        strncpy(temp, expression, open_pos - expression);
        temp[open_pos - expression] = '\0';
        strcat(temp, inner_result);
        strcat(temp, close_pos + 1);

        strcpy(expression, temp);
    }

    process_operation(expression, result);
}
```

- strrchr로 문자열 가장 안 쪽에 있는 '(' 와 ')' 를 찾는다.
- 내부 식을 추출하여 계산을 진행한다.
- 결괏값을 다시 입력값에 저장하여 다시 계산한다.

<문제>

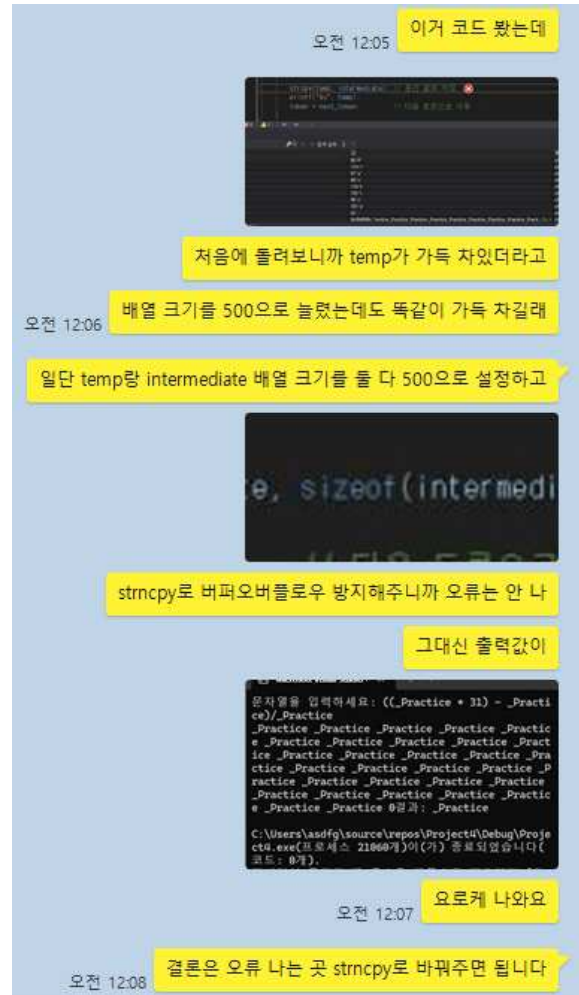


- ((\_Practice\*31) - \_Practice) / \_Practice 를 입력값으로 넣으면 곱셈 연산이 끝난 뒤에 메모리 오류가 발생하였다.



### <문제 해결 과정>

- 팀장과 조원들의 피드백을 통해 문제를 해결하였다.
- b 배열의 크기와 **strcpy**의 사용으로 인한 버퍼 오버플로우 문제였음을 파악하고 b의 배열 크기를 늘리고 **strncpy**로 대체하였다.



### <해결책을 적용하여 strncpy를 사용한 코드>

```
char inner[1000] = "";
strncpy(inner, open_pos + 1, close_pos - open_pos - 1);
inner[close_pos - open_pos - 1] = '\\0';
```

### 3. 최종 다항 연산 구조 분석

#### 1) 전체 설계 개요

해당 프로그램은 사용자가 입력한 문자열 수식을 계산하는 문자열 계산기로, 전체적인 구조는 아래와 같다.

#### 가) 프로그램 흐름

- 문자열을 입력받는다.

```
//문자열 입력받기
printf("문자열을 입력하세요: ");
gets(sentence);
```

- 괄호 여부를 확인하고 괄호의 수를 센다.

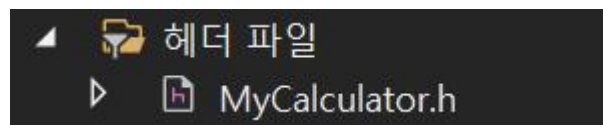
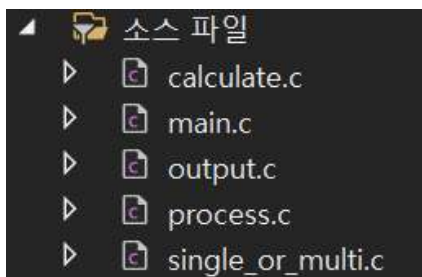
```
//sentence에서 (의 수를 세어서 while문의 조건을 생성.
int paren = 0;
count_paren(sentence, &paren);
```

- 괄호의 수가 0개라면 이항 계산을, 0개보다 크다면 다항 계산을 수행한다.

```
//CALCULATE 매크로를 활용하여 이항/다항 계산 수행.
CALCULATE(paren, sentence, result, my_op, op, &time, &count);
```

#### 나) 코드 설계

- 분할 컴파일러를 통해 소스코드와 헤더 파일을 분리하였다.



- 아래는 헤더 파일의 일부이다. 공통 전처리문을 헤더 파일에 배치하고, 각 소스코드에서는 `#include "MyCalculator.h"`만 추가하여 코드의 가독성을 높이고 수정이 쉽도록 개선하였다.

```
#ifndef MYCALCULATOR_H
#define MYCALCULATOR_H

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>

#define SIZE 500 //문자열 크기
#define CALCULATE(paren, sentence, result, my_op, op, time, count) #
    if ((paren) == 0) #
        single_calculate(sentence, result, my_op, op, time, count); #
    else if ((paren) > 0) #
        multi_calculate(sentence, result, my_op, op, paren);
```

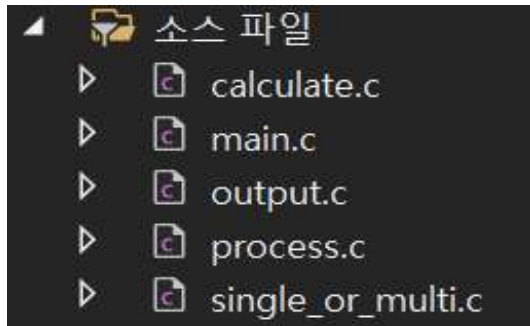
문자열 배열의 크기는 항상 500으로 지정하는 매크로 SIZE와, 괄호의 개수에 따라 이항/다항 연산을 선택하는 조건부 연산자 매크로 CALCULATE를 정의하였다.

- 그 밑으로는 함수를 선언하였다.

```
//함수 선언
void count_paren(char* sentence, int* paren);
void single_calculate(char* sentence, char* result, char* my_op, char op, int* time, int* count);
void multi_calculate(char* sentence, char* result, char* my_op, char op, int paren);
void find_op(char* sentence, char* op);
void find_section(char* sentence, char* section);
void calculate(char* pre_sen, char* next_sen, char op, char* result, int* time, int* count);
void my_add(char* a, char* b, char* result);
void my_sub(char* a, char* b, char* result);
void my_multiply(char* a, int b, char* result);
void my_divide(char* a, char* b, int* count);
void new_sentence(char* sentence, char* section, char* op, char* result, int* count);
void middle_output(char* sentence, int count, char op, int i);
void final_output(char* result, int count, char op);

#endif
```

- 소스코드 파일의 수가 너무 많아지는 것을 방지하기 위하여 관련된 함수 몇 개를 묶어 총 5개의 소스코드 파일로 분할 컴파일하였다.



1. main.c

: 문자열을 입력받고, 괄호의 개수를 세고 이항/다항 계산을 수행하는 함수를 호출한다.

2. process.c

: 문자열을 입력받고 계산에 이르기까지의 중간 과정을 수행한다. 연산기호를 찾는 함수, 가장 먼저 계산할 계산식을 찾는 함수, 새로운 문자열을 재구성하는 함수가 들어가 있다.

3. single\_or\_multi.c

: 괄호 수를 세고, 수에 따라 이항/다항 계산을 수행한다.

4. calculate.c

: 계산을 수행하는 함수(my\_add, my\_sub 등)가 포함 되어있다.

5. output.c

: 중간/최종 결과를 출력한다.

## 2) 주요 기능 및 알고리즘 설명

### ■ 주요 기능 설명

#### 가) process.c

문자열을 입력받고 계산에 이르기까지의 중간 과정을 수행한다.

- `void find_op(char* sentence, char* op):` 연산 기호를 찾는다.

```
//연산기호 찾기
void find_op(char* sentence, char* op) {
    char op_list[] = "+-*/";
    int i = 0;

    do {
        char* pos = strchr(sentence, op_list[i]);
        if (pos != NULL) {
            *op = op_list[i];
            break;
        }
        i++;
    } while (i < sizeof(op_list) - 1);
}
```

초기 이항 연산 코드와 동일하다.

- `void find_section(char* sentence, char* section)`  
: 괄호 내 가장 먼저 처리해야 할 계산식을 찾는다.

```
//괄호 내 가장 먼저 처리해야 할 섹션 찾기
void find_section(char* sentence, char* section) {
    char save[SIZE][SIZE];
    int index = 0;

    //괄호를 기준으로 나누고 저장하기
    char* token = strtok(sentence, "()");
    while (token != NULL) {
        strcpy(save[index], token); //계산용

        //저장된 save의 내용 앞뒤 공백 제거 => ( programming + c ) ... (programming + c)
        char* front = save[index];
        while (*front == ' ') front++;
        strcpy(save[index], front);

        char* back = save[index] + strlen(save[index]) - 1;
        while (*back == ' ') {
            *back = '\0';
            back--;
        }

        index++;

        token = strtok(NULL, "()");
    }
}
```

- `strtok`를 사용해 괄호를 기준으로 문자열을 나눈다.
- 나눈 문자열을 `save`(이차원 문자형 배열)에 저장한다.
- `save`에 저장된 문자열의 앞, 뒤 공백을 제거한다.  
ex) { programming + c } => {programming + c}

```
//가장 먼저 계산할 섹션 탐색
for (int i = 0; i < index; i++) {
    char op_list[] = "+-*/";

    char first = save[i][0]; //save의 첫 번째 문자
    char last = save[i][strlen(save[i]) - 1]; //save의 마지막 문자

    //양쪽 끝 문자가 연산 기호인지 확인하기
    int j = 0;
    int condition = 1; //section에 저장할 조건(condition)
    do {
        //save의 양끝에 모두 연산기호가 없어야 하므로 역을 이용
        if (first == op_list[j] || last == op_list[j]) {
            condition = 0; //만약 if문을 충족시켰다면, 우선순위에 해당하는 조건을 만족시키지 못함.
            break;
        }
        j++;
    } while (op_list[j] != '\0'); //+-*/ 다 비교하기

    //조건을 충족시켰다면 section에 저장
    if (condition) strcpy(section, save[i]);
}
}
```

- 가장 먼저 계산되는 이항식의 양 끝에 연산기호가 없다는 특징을 살려 계산 우선 순위를 지정하였다.
- 위의 조건과 일치하면 `strcpy`를 사용해 `section`(첫 번째 계산식)에 해당 식을 저장한다.

- `void new_sentence(char* sentence, char* section, char* my_op, char* result, int* count)`  
: `sentence`(가장 처음 문자열)를 재구성한다.

```
//sentence 재구성 함수
void new_sentence(char* sentence, char* section, char* my_op, char* result, int* count) {
    char a[SIZE] = ""; //section 이전 문자열
    char b[SIZE] = ""; //section 이후 문자열

    //섹션 위치 찾기
    char* start = strstr(sentence, section);

    //만약 sentence에서 섹션의 시작주소를 찾았고, 그 앞의 문자가 '('라면 ...
    if ((start != NULL) && (*(start - 1) == '(')) {
        char* end = start + strlen(section); //섹션 끝 위치
        if (*end == ')') { //')' 포함 여부 확인
            start--; //section에 '(' 포함
            end++;   //section에 ')' 포함
        }

        //섹션 이전 부분 복사
        strncpy(a, sentence, start - sentence);
        a[start - sentence] = '\0';

        //섹션 이후 부분 복사
        strcpy(b, end);

        //sentence 재구성
        strcpy(sentence, a); //섹션 이전 부분
        if (my_op == "/") strcat(sentence, count); //결과 문자열
        else strcat(sentence, result); //결과 문자열
        strcat(sentence, b); //섹션 이후 부분
    }
}
```

- 연산 우선 순위를 지정하는 조건을 반복해서 사용할 수 있도록 `sentence`(첫 번째 문자열)에서 계산한 식을 그의 결괏값으로 대체하여 새로운 문자열을 만들고, 다시 연산 우선 순위를 정하여 계산하는 방식을 사용하였다.

ex)  $((\_Practice * 3) - \_Practice) / \_Practice$   
 $\rightarrow ((\_Practice \_Practice \_Practice) - \_Practice) / \_Practice$   
 $\rightarrow \_Practice \_Practice / \_Practice$   
 $\rightarrow 2$



나) single\_or\_multi.c

- `void count_paren(char* sentence, int* paren)`  
: 문자열에서 '('의 개수를 셸다.

```
// '(' 개수 세기
void count_paren(char* sentence, int* paren) {
    for (int i = 0; i < strlen(sentence); i++) {
        if (sentence[i] == '(') (*paren)++;
    }
}
```

- `strchr`을 사용하려 했으나, 단일 문자끼리 비교하는 코드가 더 간단해 해당 조건식을 선택하였다.
- `sentence`(처음 문자열)의 단일 문자를 `for`문을 통해 하나하나 비교하여, '('와 같다면 `*paren`(괄호의 개수)을 1씩 증가시킨다.
- `void single_calculate(char* sentence, char* result, char* my_op, char op, int* time, int* count)`  
: 이항 연산을 수행한다.  
\* 참고

```
#define CALCULATE(paren, sentence, result, my_op, op, time, count) \
    if ((paren) == 0) \
        single_calculate(sentence, result, my_op, op, time, count); \
    else if ((paren) > 0) \
        multi_calculate(sentence, result, my_op, op, paren);
```

```
//이항 계산
void single_calculate(char* sentence, char* result, char* my_op, char op, int* time, int* count) {
    //연산 기호 찾기
    find_op(sentence, &op);

    //연산 기호 문자열화
    my_op[0] = op;
    my_op[1] = '\0';

    char a[SIZE] = "";
    char b[SIZE] = "";

    //연산자 기준으로 섹션 나누기
    char* token = strtok(sentence, my_op);
    strcpy(a, token);
    token = strtok(NULL, my_op);
    strcpy(b, token);

    char my_op[2] = ""; // 연산 기호 문자열로 변경
```

- 문자열에서 연산기호를 찾고, 이를 기준으로 `a`(피연산 문자열)과 `b`(연산 문자열)을 나누기 위해 `my_op`(연산기호 문자열화)를 선언한다.

```
//공백 제거
char* pos = a;
while (*pos == ' ') pos++;
strcpy(a, pos);

pos = b;
while (*pos == ' ') pos++;
strcpy(b, pos);

//계산 수행
calculate(a, b, op, result, &time, &count);

//최종 결과 출력
final_output(result, count, op);
```

- a(피연산 문자열)과 b(연산 문자열)의 앞 부분 공백을 제거하여 문자 간 공백이 하나만 있도록 설정한다.
- 이를 토대로 계산을 수행한다.
- 계산의 결과값을 연산기호에 따라 출력한다.
- `void multi_calculate(char* sentence, char* result, char* my_op, char op, int paren)`  
: 다항 연산을 수행한다.

```
//다항 계산
void multi_calculate(char* sentence, char* result, char* my_op, char op, int paren) {
    char section[SIZE]; // 섹션 배열
    char temp[SIZE]; // 임시 배열
    int time = 0; // 곱셈 횟수
    int count = 0; // 나눗셈 결과

    for (int i = 0; i < paren + 1; i++) {

        strcpy(section, ""); // section 초기화
        strcpy(my_op, ""); // my_op 초기화

        strcpy(temp, sentence); //임시 배열에 맨 처음 sentence 복사

        find_section(sentence, section); //섹션 찾기

        strcpy(sentence, temp); //바뀐 sentence에 temp(처음 sentence 복사본) 복사
        strcpy(temp, section); //임시 배열에 section 복사

        //연산 기호 찾기
        find_op(section, &op);

        //연산 기호 문자열화
        my_op[0] = op;
        my_op[1] = '\0';
    }
}
```

- `section`(연산할 문자열)과 `my_op`(문자열 연산기호)를 초기화하여 연산이 안전하게 수행될 수 있도록 한다.
- 문자열 배치 전 후에 `temp`(임시 문자형 배열)에 바뀌기 전 문자열을 복사하여 연산에 문제가 없도록 한다.  
ex) sentence = "Practice\_everyday"  
char\* tok = strstr(sentence, "tice") → sentence = "tice\_everyday"  
위와 같이 바뀌는 문제를 방지하기 위해 임시 배열을 사용하여 원래 문자열로 돌려놓는다.
- 연산기호를 찾고 문자열화 한다.



```
if (op != 'W0') {
    char a[SIZE] = "";
    char b[SIZE] = "";

    //연산자 기준으로 섹션 나누기
    char* token = strtok(section, my_op);
    strcpy(a, token);
    token = strtok(NULL, my_op);
    strcpy(b, token);

    strcpy(section, temp); // 섹션 복구
}
```

- 연산기호를 찾았다면 **strtok**를 사용하여 sentence를 my\_op 기준으로 나눈다.
- 나눈 각 문자열을 배열 a, b에 저장한다.

```
//공백 제거
char* pos = a;
while (*pos == ' ') pos++;
strcpy(a, pos);

pos = b;
while (*pos == ' ') pos++;
strcpy(b, pos);

//계산 수행
calculate(a, b, op, result, &time, &count);

//연산 기호에 따른 sentence 재구성
new_sentence(sentence, section, my_op, result, &count);
```

- a(피연산문자열)과 b(연산 문자열)의 앞 부분 공백을 제거하여 문자열 간의 공백이 하나만 있도록 설정한다.
- 이를 바탕으로 계산을 수행한다.
- 결과값을 바탕으로 새로운 문장을 재구성한다.

```
//중간 결과 출력
middle_output(sentence, count, op, i);
}

//최종 결과 출력
final_output(result, count, op);
}
```

- 문자열 내의 paren(괄호의 수) + 1 만큼 해당 내용을 반복하고, 반복할 때마다 중간 결과를 출력한다. 반복문이 끝났다면 최종 결과를 출력한다.

## 다) calculate.c

해당 소스코드의 내용은 초기 이항 연산과 동일하므로 자세한 설명은 생략하겠다.

```
//계산 수행 함수
void calculate(char* pre_sen, char* next_sen, char op, char* result, int* time, int* count) {
    if (op == '+') {
        my_add(pre_sen, next_sen, result);
    }
    else if (op == '-') {
        my_sub(pre_sen, next_sen, result);
    }
    else if (op == '*') {
        //time(곱하는 횟수)을 a의 배열값과 아스키코드 연산을 활용하여 int형으로 변환해줌
        int i = 0;
        if (next_sen != NULL && *next_sen != ' ') {
            i = atoi(next_sen);
        }
        *time = i;
        my_multiply(pre_sen, *time, result);
    }
    else if (op == '/') {
        my_divide(pre_sen, next_sen, count);
    }
}
```

- 곱셈의 경우 연산 문자열이 정수형 변수로 변환해야만 하는 문제가 있어, chap 10에 나오는 atoi 함수를 사용했다.
- stdlib.h에 원형 정의 반드시 포함

함수	설명
int atoi( const char *str );	str을 int형으로 변환한다.
long atoi( const char *str );	str을 long형으로 변환한다.
double atof( const char *str );	str을 double형으로 변환한다.

라) output.c

- `void middle_output(char* sentence, int count, char op, int i)`  
: 중간 결과를 출력한다.

```
//중간 결과 출력
void middle_output(char* sentence, int count, char op, int i) {
    //괄호 처리가 아직 남았다면 (어차피 문자열이 남아있으니)
    if (strchr(sentence, '(') != NULL) {
        printf("%d차 중간 결과: %s\n", i + 1, sentence);
    }

    //괄호 처리가 다 끝났다면 (숫자만 남았을 경우를 대비하여)
    else {
        if (op == '/') printf("%d차 중간 결과: %d\n", i + 1, count);
        else printf("%d차 중간 결과: %s\n", i + 1, sentence);
    }
}
```

- 문장 내의 괄호가 남아있는지, 연산기호가 무엇인지를 통해서 출력값의 종류를 결정한다.
- 이는 정수가 출력되는 나눗셈의 경우를 고려한 것이다.

- `void final_output(char* result, int count, char op)`  
: 최종 결과를 출력한다.

```
//최종 결과 출력
void final_output(char* result, int count, char op) {
    if (op == '/') printf("최종 결과: %d\n", count);
    else printf("최종 결과: %s\n", result);
}
```

- 최종 결과의 경우, 마지막 연산이 나눗셈이면 정수를 출력해야 하므로 if~else를 통해 출력 종류를 나누어 주었다.

#### 4. 최종 다항 연산 디버깅 및 결과 분석

이름	값	형식
count	0	int
my_op	0x0000004010fbfb4 ""	char[2]
op	0 '\0'	char
paren	-858993460	int
result	0x0000004010fbfb80 ""	char[500]
section	0x0000004010fbfb20 ""	char[500]
sentence	0x0000004010fbfb60 ""	char[500]
temp	0x0000004010fbfb240 ""	char[500]
time	0	int

- 필요한 지역 변수를 선언한다.

```
//문자열 입력받기
printf("문자열을 입력하세요: ");
gets(sentence);
```

문자열을 입력하세요: (( \_Practice \* 31) - \_Practice) / \_Practice

▶ sentence	0x0000004010fbfb60 "(( _Practice * 31) - _Practice) / _Practice"	char[500]
------------	--	-----------

- 초기 문자열을 입력한다.

```
//sentence에서 (의 수를 세어서 while문의 조건을 생성
int paren = 0;
count_paren(sentence, &paren); 경과 시간 1ms 이하
```




- 문자열 내 '('의 수를 0으로 초기화한 뒤, count\_paren 함수를 실행한다.

▶ paren	0x0000004010fbf454 {2}
▶ sentence	0x0000004010fbfb60 "(( _Practice * 31) - _Practice) / _Practice"


- '('의 수가 2인 것을 알 수 있다.

```
//CALCULATE 매크로를 활용하여 이항/다항 계산 수행.
CALCULATE(paren, sentence, result, my_op, op, &time, &count);
```




- CALCULATE 매크로를 활용하여 이항/다항 연산을 수행한다.

▶  sentence	0x0000004010fbeb60 "(_Practice * 31"	Q ▾	char *
▶  strtok	0x00007ff7df7a6fbc {MyCalculator.exe!strtok}		void *
▶  token	0x0000004010fbeb62 "_Practice * 31"	Q ▾	char *






- 괄호를 기준으로 가장 앞쪽의 계산식을 분리하여 token에 저장한다.

▶  save[index]	0x0000004010f805e0 "_Practice * 31"
---	-------------------------------------

- token이 NULL이 아니라면 이차원 배열 save에 저장한다.
- 이때 문자열의 앞뒤 공백을 제거한다.

▶  sentence	0x0000004010fbeb60 "(_Practice * 31"	Q ▾	char *
▶  strtok	0x00007ff7df7a6fbc {MyCalculator.exe!strtok}		void *
▶  token	0x0000004010fbeb71 " - _Practice"	Q ▾	char *

- 다음으로 분리된 문자를 token에 저장한다.
- token이 NULL이 아니라면 위의 동작을 반복한다.

▶  save	0x0000004010f805e0 {0x0000004010f805e0, 0x0000004010f807d4, 0x0000004010f809c8, 0x0000004010f80bbc}
▶  [0]	0x0000004010f805e0 "_Practice * 31"
▶  [1]	0x0000004010f807d4 " - _Practice"
▶  [2]	0x0000004010f809c8 " / _Practice"
▶  [3]	0x0000004010f80bbc "ㄷㄷㄷㄷㄷㄷㄷ"

- 위와 같이 분리된 문자열들이 이차원 배열 save에 저장된 것을 볼 수 있다.

```
char op_list[] = "+-*/"; 경과 시간 1ms 이하

char first = save[i][0]; //save의 첫 번째 문자
char last = save[i][strlen(save[i]) - 1]; //save의 마지막 문자
```

- 연산기호 배열인 op\_list를 선언하고, save의 i번째 문자열의 첫 번째 문자와 마지막 문자를 문자형 변수 first와 last에 저장한다.

```
//양쪽 끝 문자가 연산 기호인지 확인하기
int j = 0;
int condition = 1; //section에 저장할 조건(condition)
do {
    //save의 양끝에 모두 연산기호가 없어야 하므로 역을 이용
    if (first == op_list[j] || last == op_list[j]) {
        condition = 0; //만약 if문을 충족시켰다면, 우선순위에 해당하는 조건을 만족시키지 못함.
        break;
    }
    j++;
} while (op_list[j] != '\0'); //+-*/ 다 비교하기
```

- '첫 번째 문자와 마지막 문자가 모두 연산기호가 아니다'의 역을 조건으로 두어 가장 먼저 계산될 계산식을 정한다.
- 이때 condition이 1이면 가장 먼저 계산될 조건을 충족하는 것이고, 0이면 충족하지 못하는 것이다.

```
//조건을 충족시켰다면 section에 저장
if (condition) strcpy(section, save[i]); 경과 시간 1ms 이하
```

section	0x0000004010fbde30 "_Practice * 31"	char *
---------	-------------------------------------	--------

- 이렇게 가장 먼저 계산될 계산식 배열 section에 가장 안쪽 괄호의 문자열이 저장된다.

my_op	0x0000004010fbefb4 "+*"
op	42 '+'

- section(이항 연산식)에서 연산기호를 찾고 문자열화 한다.



a	0x0000004010fbe2d0 "_Practice"	Q	char[500]
b	0x0000004010fbe4f0 " 31"	Q	char[500]

- 문자형 배열 a와 b를 선언하고 피연산문자열과 연산문자열을 각각 저장한다.

a	0x0000004010fbe2d0 "_Practice"	Q	char[500]
b	0x0000004010fbe4f0 "31"	Q	char[500]

- 각 문자열 앞의 공백을 제거한 모습이다.

i	31	int
next_sen	0x0000004010fbe4f0 "31"	char *

- 이를 토대로 연산문자열(b)를 atoi함수를 사용해 정수형 변수로 변환한다.

[illegible]

- \_Practice가 31번 곱해진 결과값이다.

▶ start	0x0000004010fbefb61 "(Practice * 31) - Practice) / Practice"
▶ end	0x0000004010fbefb71 "- Practice) / Practice"

- 기존 문자열을 재구성하기 위해서 section의 앞과 뒤를 가리키는 포인터에 앞뒤 괄호를 포함한다.

```
문자열을 입력하세요: ((_Practice * 31) - _Practice) / _Practice
1차 중간 결과: (_Practice _Practice _Practice _Practice _Practice _Practice
_Practice _Practice _Practice _Practice _Practice _Practice _Practice _Pra
ctice _Practice _Practice _Practice _Practice _Practice _Practice _Practice
_Practice _Practice _Practice _Practice _Practice _Practice _Practice - _Practice)
/ _Practice
```

- 이를 토대로 strcpy, strcat을 활용하여 새로운 문자열을 구성하고 중간 결과를 출력한다.

```
2차 중간 결과: _Practice _Practice _Practice _Practice _Practice _Practice _Practice
_Practice _Practice _Practice _Practice _Practice _Practice _Practice _Practice _Pr
actice _Practice _Practice _Practice _Practice _Practice _Practice _Practice _Practice
_Practice _Practice _Practice _Practice _Practice _Practice / _Practice
```

```
3차 중간 결과: 30
```

```
최종 결과: 30
```

- 해당 내용을 (괄호의 수 + 1) 만큼 반복한다.

```
문자열을 입력하세요: ((_Practice * 31) - _Practice) / _Practice
1차 중간 결과: (_Practice _Practice _Practice _Practice _Practice _Practice _Practice
_Practice _Practice _Practice _Practice _Practice _Practice _Practice _Practice _Pr
actice _Practice _Practice _Practice _Practice _Practice _Practice _Practice _Pra
actice _Practice _Practice _Practice _Practice _Practice _Practice _Practice - _Practice)
/ _Practice
2차 중간 결과: _Practice _Practice _Practice _Practice _Practice _Practice _Practice
_Practice _Practice _Practice _Practice _Practice _Practice _Practice _Practice _Pr
actice _Practice _Practice _Practice _Practice _Practice _Practice _Practice _Pra
actice _Practice _Practice _Practice _Practice _Practice _Practice _Practice _Practice
_Practice _Practice _Practice _Practice _Practice _Practice / _Practice
3차 중간 결과: 30
최종 결과: 30
```

- 최종 출력값이다.



### III. 결론

#### 1. 느낀 점 및 학습 내용

##### 김나연(팀장, 스마트보안전공 202435143)

혼자서 하는 과제보다 팀원들이 받쳐주는 팀 프로젝트가 더 완성도 있는 결과물을 도출 해낼 수 있었다. 질문을 받고 답을 주는 과정에서 공부도 되었고, 내가 모르는 것을 팀원에게 질문하며 몰랐던 것을 배움으로써 팀 프로젝트의 장점을 느낀 좋은 경험이 되었다고 생각한다. 또한 팀장이라는 직책을 맡으며 어떻게 팀을 이끌어야 하는지에 대한 고민과, 진행 과정에서의 어려움을 해결하기 위해 노력했던 고민이 새로운 경험으로써 한 단계 성장하는 계기가 되었으며, 나뿐만 아니라 한 단계 성장한 팀원들의 모습에 뿌듯함도 느꼈다.

##### 박이진(스마트시티융합학과, 202335230)

내가 해결하지 못한 문제점을 다함께 공유해서 빨리 해결할 수 있었고 한 문제를 해결할 때 생각하지 못한 여러 가지 의견과 방법을 볼 수 있었다. 각자 자신의 의견을 자유롭게 냈던 것 같고 프로젝트 진행이나 회의를 하는데 있어서 팀장님의 역할이 정말 컸다고 생각한다.

##### 오시현(스마트보안전공, 202435166)

이 문자열 계산기 프로젝트를 진행하면서 지금까지 배워왔던 모든 함수를 다 활용해본 것 같아 실력이 많이 오른 것 같다. 또, 내 아이디어로만 프로젝트를 진행했던 첫 번째 과제와 다르게 이번 팀 프로젝트에서는 팀원들이 여러 아이디어를 들고 여러 구현 방식에 대해 고민해 볼 수 있어서 좋았고 각자 구현이 완료된 후 코드를 비교해 보면서 어떤 게 더 보기 편하고 간단할지 분석해보니 "프로그래밍을 하는 스타일은 정말 여러 개가 있을 수 있구나"라고 생각했다.

##### 김현아(스마트보안전공, 202435158)

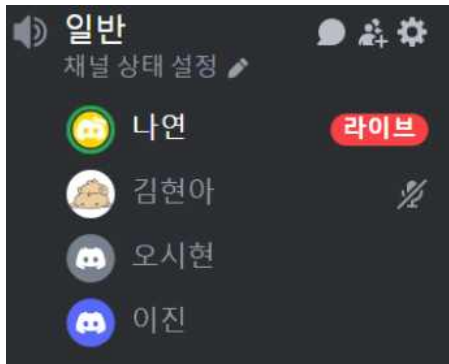
팀프로젝트의 장점을 알았다. 팀끼리 회의를 하면 여러 아이디어가 나와서 어떤 방식으로 구현할지 정하는 것이 굉장히 수월하였다. 또한 나누어서 코드를 짜면 방대한 양의 코드도 빠른 시간 안에 짤 수 있던 점과, 모르는 것이 있을 때 다들 비슷한 오류를 겪었거나 배운 내용을 활용할 수 있기에 빠른 피드백이 가능했던 점이 좋았다.

#### 2. 향후 개선 방안

- 1) 괄호가 없는 다항 연산을 구현함으로써 계산기의 정확도를 개선하고 싶다.  
ex) `practice*3-practice >> practice practice` (지금 코드에선 구현 안됨.)
- 2) 단항연산 결과는 1차 중간 결과로 끝난다는 점을 개선하고 싶다.
- 3) 괄호와 문자열 사이에 공백이 있는 경우 연산이 수행되지 않는 것을 개선하고 싶다.

## IV. 부록

### 1. 회의 기록



온라인 회의 모습

< 20241109 회의록 >

- 구조체 사용 여부  
다항 구현 이후에 유동적으로 추가할 것.  
(다만 시간이 오래 걸릴 경우 생략하고 보고서에 더 힘을 쏟기)
- 다항 구현 방법  
각자 취합한 코드를 바탕으로 \*다음주 목요일(11/14)\* 회의 전까지 구현하여 카카오휴크에 업로드 할 것.  
(구현 중에 자신이 맡은 부분에서 수정이 필요할 경우 언제든지 카카오휴크에 피드백 남겨주세요)
- 문장 입출력 방식  
(문자열) (연산기호) (문자열) 형식으로 입력받을 것.

\*질문이나 논의할 사항이 있으시다면 언제든지 카카오휴크에 남겨주세요.)

오후 5:25

2024년 11월 12일 화요일

문자열에서 단어를 \_로 이은 것은 덧셈, 곱셈에서 단어가 나열될 때 서로 붙어서 알아보기 어려운 경우를 방지하기 위함이라고 합니다  
ex) tree+flower = treeflower

저희 조는 \_가 아닌 공백으로 이를 대신했으니 괜찮다고 하셨습니다.)

오후 10:11

늦은 시간 메시지 죄송합니다  
구현 중에 특별히 어렵거나, 함께 공유하고 싶은 오류가 있다면 캡처해서 디스코드 #오류 채널에 업로드해주세요!  
보고서에 이 부분을 추가해야 할 것 같아서 공지 드립니다 😊

오후 11:31

< 20241115 회의록 >

- 중괄호, 대괄호는 제외하고 소괄호만 고려할 것
- 현재 진행 상황까지의 코드를 간단하게 설명해서 업로드 할 것  
(보고서 작성에 필요하니, 오류가 나는 부분과 새로 구현한 부분을 디버깅 및 캡처해서 설명하는 한글/워드 파일을 올려주세요. 다항연산 구현 부분을 위주로 작성해주세요.)
- \*다음주 화요일(11/19)\* 회의 시간 전까지 구현과 업로드를 완료할 것  
(시간이 촉박할 것 같다면 언제든지 괜찮으니 카카오휴크로 얘기해주세요)

오후 1:22

각자의 코드 설명 업로드는 조원이 4명이면 아이디어도 4개가 나와야 한다는 교수님의 강조가 있었기에 진행하는 것입니다.  
그래서 너무 비슷한 경우에는 조금씩 수정할 계획도 있습니다.  
귀찮으시겠지만 꼭 업로드해주세요 😊

오후 1:25

오시현

소스.c  
6.32 KB  
저장 · 다른 이름으로 저장

소스코드 파일 먼저 올리겠습니다!

오후 1:28

박이진

123.c  
8.31 KB  
저장 · 다른 이름으로 저장

c팀플\_박이진.docx  
130.82 KB

2024년 11월 19일 화요일

오늘 11-2시 팀들에는 자신의 다항구현 방법을 간략하게 설명한 뒤, 가장 효율적인 코드를 최종 다항연산 구현 방법으로 채택할 예정입니다.

이후에는 레포트를 작성할 계획입니다. 다같이 레포트를 작성하는 것을 계획했으나 이는 비효율적이라 판단되어 이번 주 금요일까지 한 명이 초안을 작성한 뒤 제출 전까지 레포트 피드백을 진행할 것입니다.

오전 8:29

어제 현아님의 코드를 보니 제 코드와 구현 방식에 있어서 비슷한 점이 많아 저는 strtok를 사용하는 방식으로 변경하고 있습니다. 아직 현아님도 코드를 완성하지 못 하셨구요.

고로 오늘은 번갈아 오시는 시현님과 이진님이 >기본적인< 레포트의 틀을 만들어주시면 될 것 같습니다.

오전 8:32

오시현

넵 알겠습니다



오전 8:41



< 공지 >

- 오늘(11/19)까지 아이디어 제출
  - 다항 연산 구현에 대한 아이디어를 내주세요.  
(순전히 본인만의 아이디어를 원합니다 겹쳐도 괜찮으니 찻지피티는 사용하지 말아주세요)
- 코드 구현은 목요일(11/22) 오후 6시까지 해서 제출
  - 이건 날짜만 알고 계세요. 아이디어 보고 다시 공지하겠습니다
- 보고서는 금-토 동안 완성할 예정
  - 채택된 코드의 제작자가 초안을 작성할 것입니다.  
다른 팀원들은 그에 대한 피드백을 작성해서 레포트에 추가하는 방식으로 진행하겠습니다.

=====

< 추가해야 할 부분 >

- 조건부 컴파일러 활용하기
- 소스파일과 헤더파일 따로따로 구성 하기
  - 채택된 아이디어를 구성하는 팀원 외의 다른 팀원들은 chap 14에 대한 공부 후에, 완성된 코드에 대한 구성을 변경할 것.

< 공지 >

1. 오늘(11/19)까지 아이디어 제출

- 다항 연산 구현에 대한 아이디어를 내주세요.  
(순전히 본인만의 아이디어를 원합니다 겹쳐도 괜찮으니 찻지피티는 사용하지 말아주세요)

2. 코드 구현은 목요일(11/22) 오후 6시까지 해서 제출

- 이건 날짜만 알고 계세요. 아이디어 보고 다시 공지하겠습니다

3. 보고서는 금-토 동안 완성할 예정

- 채택된 코드의 제작자가 초안을 작성할 것입니다.  
다른 팀원들은 그에 대한 피드백을 작성해서 레포트에 추가하는 방식으로 진행하겠습니다.

< 추가해야 할 부분 >

- 조건부 컴파일러 활용하기
- 소스파일과 헤더파일 따로따로 구성 하기
  - 채택된 아이디어를 구성하는 팀원 외의 다른 팀원들은 chap 14에 대한 공부 후에, 완성된 코드에 대한 구성을 변경할 것.

< 20241123 회의록 >

1. 오늘에서 일요일 오후 중으로 보고서 완성해서 보내드릴 테니 꼭 꼼꼼히 보시고 이상한 부분 있으면 말씀해주세요!

2. 각자 코드에서 발생했던 문제들과 해결 과정을 적어주세요. 사진을 찍어서 보내주시면 더 좋습니다 :)

부족한 점이 많은 팀장이었지만, 끝까지 믿고 잘 따라와 주셔서 정말 감사드립니다. 모두 정말 수고 많으셨어요! 문제 생기면 다시 카카오톡으로 연락 드리겠습니다 ^\_ ^

오전 12:19

## 2. 구성원의 역할 분담 및 기여점

### 김나연(팀장)

- 팀원 간의 역할 분담과 일정 조율, 회의 진행을 담당하였다.
- 연산 중 나눗셈 구현을 맡았다.
- 프로젝트의 진행 상황을 관리하며, 팀원들이 맡은 기능 구현을 취합하였다.
- strtok로 계산식을 나누어 연산 우선순위를 결정하는 방식의 다항 연산 프로그램을 제작하였다.
- 팀원들의 기술적 질문을 해결하며 프로젝트 진행 과정에서 원활한 협업을 도왔다.
- 전반적인 보고서 작성과 발표를 맡았다.

### 박이진

- 연산 중 곱셈 구현을 맡았다.
- 작성한 곱셈 코드의 작동 원리를 분석하고, 디버깅을 통한 코드 설명 보고서를 작성하였다.
- 매 회의에 참석하여 다항 연산 구현 방식에 대한 의견을 나누고, 각 연산의 오류와 개선 방안에 대해 논의하였다.
- 재귀함수를 사용하는 방식의 다항 연산 프로그램을 제작하였다.
- 구현한 다항 연산에서 발생한 오류에 대해 팀장과 팀원에게 피드백을 요청하며, 이를 통해 코드를 개선하고 코딩 실력을 향상하였다.

### 오시현

- 연산 중 뺄셈 구현을 맡았다.
- 뺄셈과 나눗셈 구현 방식에 대해 팀장과 논의하며 strtok를 사용하여 계산식을 나누는 방식과 strstr을 사용하여 특정 단어를 검색하는 방식의 두 뺄셈 기능 코드를 제작하였다.
- 작성한 뺄셈 코드의 작동 원리를 분석하고, 디버깅을 통한 코드 설명 보고서를 작성하였다.
- 매 회의에 참석하여 자신의 작업 진행 상황을 팀원들과 공유하고, 구현 방식에 대한 의견을 적극적으로 제시하였다.
- 닫는 괄호의 배치에 따라 연산의 우선순위를 지정하는 방식의 다항 연산 프로그램을 구현하였다.

### 김현아

- 연산 중 덧셈 구현을 맡았다.
- 작성한 덧셈 코드의 작동 원리를 분석하고, 디버깅을 통한 코드 설명 보고서를 작성하였다.
- 매 회의에 참석하여 프로젝트 진행에 이바지하였으며, 진행 방식에 대한 의견을 적극적으로 제시함으로써 팀장이 진행 방향을 결정하는 데 실질적인 보탬이 되었다.
- 가장 안쪽에 있는 여는 괄호와 닫는 괄호를 strchr을 사용하여 연산의 우선순위를 지정하는 방식의 다항 연산 프로그램을 구현하였다.
- 구현한 다항 연산에서 발생한 오류에 대해 팀장과 팀원에게 피드백을 요청하며, 이를 통해 코드를 개선하고 코딩 실력을 향상하였다.