

# REPORT

## 교육목표

### 정보통신대학 교육목표

정보통신대학은 수요지향적 교육을 바탕으로 국제 경쟁력과 전문성 및 실용성을 갖춘 고급 정보통신 엔지니어의 양성을 목표로 하고 있다. 이를 달성하기 위한 세부 교육목표는 다음과 같다.

1. 국제적 경쟁력을 갖춘 정보통신인
2. 현장 적응 능력이 뛰어난 실용적 정보통신인
3. 기반 전문성을 갖춘 발전적 정보통신인
4. 윤리의식과 문화적 소양을 갖춘 정보통신인

### 전자공학 프로그램 교육목표

1. 공학 기초지식과 전문지식을 활용하여 전자공학의 시스템, 부품, 공정, 방법을 분석하고 설계하는 능력을 기른다.
2. 상호 이해와 협력, 일에 대한 분석과 기획을 통하여 복합학제적 문제를 해결하는 능력을 기른다.
3. 사회와 문화에 대한 이해 및 외국어 능력을 바탕으로 국제적으로 협조하여 일할 수 있는 엔지니어로 성장시킨다.
4. 건전한 윤리의식과 지속적 자기개발 능력을 함양하여 사회적 책임을 다하는 엔지니어로 성장시킨다.

나는 위 교육목표를 숙지하여 공학교육인증을 이수하는데 최선을 다할 것을 서약합니다.

학 부: 전자공학부

제출일: 2019.03.30

과목명: 임베디드시스템설계

학 번: 201420703

성 명: 이 상구

## ■ 문제의 조건과 필요조건 제시

아래의 Directory 구조를 참고하여, 필요조건이 반드시 충족 되도록 각 문제 1,2의 내용을 수행하는 Makefile, 이에 필요한 c 파일 또는 header 파일을 작성하시오.

조건	Directory 구조
<p>① 폴더별 각각 실행파일과 object 파일을 만들고 삭제(clean)할 수 있는 Makefile을 만드시오. 또, 각 Makefile은 Macro와 확장자 규칙을 사용하시오.</p> <p>② 각 Directory에 대해 최소 3개 이상의 c파일을 해당하는 폴더에 만들고 각 c파일에 하나 이상의 함수를 구현하시오.</p>	<pre> graph TD     homework1 --&gt; part1     homework1 --&gt; part2     part2 --&gt; part3         </pre>

## ■ 문제 1

구성도
<pre> Lee@lee-14ZD970-GX30K:~/homework\$ ls . part1 part2 part2/part3 .: Makefile  func0.c  main.c  part1  part2  read_from_shm.c  part1: Makefile  func1.c  main.c  printarr.c  part2: Makefile  extra.h  func2.c  main.c  part3  write_to_shm.c  part2/part3: Makefile  common_divisor.c  common_multiple.c  find_min.c  main.c  my_numeric.h Lee@lee-14ZD970-GX30K:~/homework\$         </pre>

홍 (~) 예하에는, homework 디렉토리가 존재하며, 이 디렉토리는 문제2에서의 main.c 함수와 나머지 함수들이 존재하며, part1 폴더에는 문제1 파트1의 함수 : func1.c main.c printarr.c 함수가 존재한다. part2 폴더에는 main.c, func2.c write\_to\_shm.c 와 헤더파일 extra.h가 존재한다. 마지막으로 part3 폴더 내부에는 main.c, common\_multiple.c, common\_divisor.c find\_min.c 함수와 my\_numeric.h 헤더파일이 존재한다. 각 디렉토리마다 Makefile이 존재하며, 이는 각 디렉토리 내의 함수들만 컴파일 해준다.

### ▶ Part 1

#### [프로그램 실행 및 출력]

조건
<p>임의의 아홉 자리의 수(학번) 입력 받아 각 자릿수 중 0을 제거하고 이 수를 출력하는 프로그램을 구현하시오.</p> <p>예) Input : 201624108 Output : 2162418</p>
<p>9개의 입력이 들어갔을 때</p> <pre> Lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework/part1\$ ./Part1.out 학번 입력: 201420703 214273  Lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework/part1\$         </pre>

9개 이하의 입력이 들어갔을 때
<pre>lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework/part1\$ ./Part1.out 학번 입력: 10201 121  lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework/part1\$ █</pre>
9개 이상의 입력이 들어갔을 때
<pre>lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework/part1\$ ./Part1.out 학번 입력: 0000110101010 111  lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework/part1\$ █</pre>

문제에서 임의의 아홉 자리의 수를 입력받는다. 이때 scanf 함수를 통하여 string 문자열을 입력받으며, 각 문자열을 검사하여 '0' 문자를 가진 문자를 제외하고 다른 배열에 복사한다.

적절히 동작하는지 확인하기 위하여, 9개의 수를 입력, 9개 이하의 수 입력, 9개 이상의 수 입력을 통하여 결과를 확인하였으며, 모두 적절히 동작함을 확인하였다. 9개 이상의 입력이 들어오게 되면, 9번째까지만 입력을 받고 나머지 입력은 무시하도록 설계하였다.

#### [소스코드 관계도]

```
part1:
Makefile func1.c main.c printarr.c
```

Part1 은 main함수를 포함하는 main.c 와 문제에서 요구하는 '0'을 제거하는 동작을 하는 핵심적인 함수가 func1.c에 작성되어있으며, 마지막으로 printarr 에 그 배열의 요소를 출력하는 함수를 구현하였다.

main.c	사용자로부터 scanf를 통하여 입력을 받으며, func1을 통하여 목표배열을 구성한다. printarr함수를 통하여 이를 출력한다.
printarr.c	문자열 배열을 출력하는 printf 함수가 포함 되어있다.
func1.c	반복문을 통하여, 문자열 s1 내의 '0' 인 문자열을 제외하고 전부 다른 배열 s2로 복사한다.

#### [소스코드 분석]

<pre>printf("학번 입력: "); //학번을 입력: scanf("%9s", in_string); // scanf로 학번을 입력받는다.</pre>
scanf을 통하여 최대 9자리까지의 문자열을 입력받는다.
<pre>for(int i = 0, j = 0 ; i &lt; _STR_LENGTH; i++ ) { //입력배열을 탐색하여      if(in_string[i] != '0') //0이 아닌이상         out_string[j++] =in_string[i]; //출력배열에 저장한다. }</pre>
in_string에 입력된 문자열을 탐색하여, in_string[i] == '0'이 되는 경우를 제외하고 모두 out_string에 동일하게 복사함을 볼 수 있다. 즉 이 과정을 통하여 '0'을 제거하게 된다.

## [Makefile 설계]

<pre>GNU nano 2.9.3  SUFFIXES : .c .o CC = gcc CFLAGS = -g  OBJS = main.o printarr.o func1.o TARGET = Part1.out  all : \$(TARGET)  \$(TARGET): \$(OBJS)     \$(CC) -o \$@ \$^  .PHONY : clean  clean:     rm -f \$(OBJS) \$(TARGET)</pre>	<p>→ .SUFFIXES 는 묵시적 확장자로서, .o 파일이 없더라도 자동적으로, .c를 통하여 .o를 컴파일한다.</p> <p>→ OBJS는 소스파일의 오브젝트 파일이름을 포함하는 매크로 변수이다. 소스코드에 main함수가 포함된 main.o 그리고 보조적인 함수 printarr, func1 등이 포함되어있다.</p> <p>→ CC를 통하여 gcc 컴파일러를 설정하였다.</p> <p>→ CFLAGS 에 -g옵션을 주어서 gdb로 디버깅을 할 수 있도록 옵션을 추가 하였다.</p> <p>→ all: 즉 Makefile이 실행하여야 할 action으로, TARGET이 선행되어야 한다. TARGET은 아래에 \$(TARGET) : 문을 통해서, \$(OBJS)를 필요로 하므로, OBJS를 찾게 된다. 이때 묵시적 .SUFFIXES에 의하여 main.c, printarr.c, func1.c 가 자동으로 main.o, printarr.o, func1.o로 컴파일되어 타겟의 선행조건, \$(OBJS) 가 충족이되어, \$(CC) -o \$(TARGET) \$(OBJS) 가 실행이 된다.</p> <p>→ 위 표현은 gcc -o Part1.out main.o, printarr.o, func1.o 와 동일하며, 성공시 Part1.out 실행이미지가 생성된다.</p>
---	--

### make 명령어 실행 전

```
part1:
Makefile func1.c main.c printarr.c
```

### make 명령어 실행 후

```
lee@lee-14ZD970-GX30K:~/homework/part1$ make
gcc -g -c -o main.o main.c
gcc -g -c -o printarr.o printarr.c
gcc -g -c -o func1.o func1.c
gcc -o Part1.out main.o printarr.o func1.o
lee@lee-14ZD970-GX30K:~/homework/part1$ ls
Makefile Part1.out func1.c func1.o main.c main.o printarr.c printarr.o
lee@lee-14ZD970-GX30K:~/homework/part1$
```

자동적으로 main.o, func1.o, printarr.o가 생성이 됨을 확인하였으며,  
Part1.out 실행파일이 생성이 되었다. 이를 실행하면,

```
lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework/part1$ ./Part1.out
학번 입력: 201420703
214273

lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework/part1$
```

와 같이 정상적으로 작동됨을 확인하였다.

## ► Part 2

### 조건

최대 아홉 자리의 수를 입력 받아 각 자릿수의 합을 출력하고 각 자릿수의 곱을 출력하는 프로그램을 구현하시오

9자리의 이하의 입력이 들어갔을 때

```
lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework/part2$ ./Part2.out
9자리의 수를 입력해주세요
21432832
합의 값: 25 곱의 값: 2304
```

#### 9개 이상의 입력이 들어갔을 때

```
lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework/part2$ ./Part2.out
9자리의 수를 입력해주세요
12345678911
합의 값: 45 곱의 값: 362880
```

9자리 또는 9자리 이하의 입력이 들어갔을 때 정상적으로 각 자리수의 합과 곱을 출력하는 것을 확인하였다. 실제로  $2 + 1 + 4 + 3 + 2 + 8 + 3 + 2$  의 경우, 25의 합을 가지며, 곱은 2304를 가짐을 계산기를 통하여 확인하였다.

하지만, 9자리 이상의 입력이 들어올 시  $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 1 + 1$  중 나머지 2개의 1은 무시되어 1~9 까지의 합인 45만 출력이 됨을 확인할 수 있으며, 마찬가지로 1~9 까지의 곱인 362,880의 값이 출력이 됨을 확인할 수 있다.

#### [소스코드 관계도]

```
part2:
Makefile extra.h func2.c main.c part3 write_to_shm.c
```

Part2 는 main함수를 포함하는 main.c 와 문제에서 요구하는 각자리의 합과 곱을 구해주는 핵심적인 함수가 func2.c에 작성되어 있다. write\_to\_shm.c 함수의 경우 문제 2번에서 구현해야 할 마름모의 가로, 높이의 길이를 다른 프로그램으로 공유메모리를 통하여 공유하는데 필요한 함수이다.

main.c	사용자로부터, 9자리의 수를 문자열로 입력받는 기능, func2함수를 호출하여, 합과 곱을 구한 후, 그 합과 곱을 write_to_shm.c을 통하여 다른 프로그램과 메모리를 공유한다.
func2.c	문자열 s1을 입력받는다. 이 입력받은 문자열은 switch 문을 통하여 '0','1'...'9'와 같이 문자단위로 비교 분석하여, sum_val, mul_val에 합과 곱을 연속해 나아가면서 저장한다. 그리고 합과 곱을 반환한다.
write_to_shm.c	문제 2, 즉 서로 다른 프로그램이 같은 값을 공유하기 위하여, ipc 방법중 공유메모리를 활용하여, part2에서 얻은 가로, 세로의 길이를, 문제2에서 구현한 프로그램이 읽을 수 있게 메모리를 공유하게 한다.

#### [소스코드 분석]

```
for(int i=0 ; i <= _STR_LENGTH_ ; i++){
//입력받은 문자열 탐색
switch(in_string[i]){ //문자 -> 정수표현으로 곱과 합을 더하거나 곱
    case '0': sum_val += 0; mul_val *= 0; break;
    case '1': sum_val += 1; mul_val *= 1; break;
    case '2': sum_val += 2; mul_val *= 2; break;
    case '3': sum_val += 3; mul_val *= 3; break;
    case '4': sum_val += 4; mul_val *= 4; break;
    case '5': sum_val += 5; mul_val *= 5; break;
    case '6': sum_val += 6; mul_val *= 6; break;
    case '7': sum_val += 7; mul_val *= 7; break;
    case '8': sum_val += 8; mul_val *= 8; break;
    case '9': sum_val += 9; mul_val *= 9; break;
    default: break;
} // end switch
} //end for
```

입력받은 문자열은 in\_string에 저장되며, 이 문자열의 인덱스를 탐색하면서 숫자문자형이 나오게 되면 그 문자열을 더하거나, 변수에 저장한다.



## [Makefile 설계]

<pre>GNU nano 2.9.3  .SUFFIXES : .c .o CC = gcc CFLAGS = -g TARGET = Part2.out  OBJS = main.o func2.o write_to_shm.o SRCS = (main:.o = .c)  all: \$(TARGET)  \$(TARGET) : \$(OBJS) extra.h             \$(CC) -o \$(TARGET) ^  clean :     rm -f \$(OBJS) \$(TARGET)</pre>	<p>→ .SUFFIXES 는 묵시적 확장자로서 .o 파일이 없더라도 자동적으로 .c를 통하여 .o를 컴파일 한다.</p> <p>→ OBJS는 소스파일의 오브젝트 파일이름을 포함하는 매크로 변수로, main함수가 존재하는 main.o 그리고 보조적인 함수 func2.o, write_to_shm.o 등이 포함되어있다.</p> <p>→ CC를 통하여 gcc 컴파일러를 설정하였다.</p> <p>CFLAGS 에 -g옵션을 주어서 gdb로 디버깅을 할 수 있도록 옵션을 추가 하였다.</p> <p>→ all: Makefile이 실행하여야 할 action으로, TARGET이 선행되어야 한다. TARGET은 아래에</p> <p><b>\$(TARGET):</b> 을 통해서, 필요한 선행조건을 찾을 수 있으며, 이때 \$(OBJS)를 필요로 하므로, OBJS를 찾게 된다. 묵시적 .SUFFIXES 매크로 변수에 의하여 main.c, func2.c, write_to_shm.c 가 자동으로 main.o, write_to_shm.o, func2.o 로 컴파일되어 타겟의 선행조건, \$(OBJS) 가 만족된다.</p> <p>이후 \$(CC) -o \$(TARGET) \$(OBJS) 가 실행이 된다.</p> <p>→ 위 표현은</p> <p>gcc -o Part2.out main.o, write_to_shm.o, func2.o extra.h 와 동일하며, 성공 시 Part2.out 실행이미지가 생성된다.</p>
--	--

### make 파일 실행 시

```
lee@lee-14ZD970-GX30K:~/homework/part2$ make
gcc -g -c -o main.o main.c
gcc -g -c -o func2.o func2.c
gcc -g -c -o write_to_shm.o write_to_shm.c
gcc -o Part2.out main.o func2.o write_to_shm.o extra.h
lee@lee-14ZD970-GX30K:~/homework/part2$ ls
Makefile Part2.out extra.h func2.c func2.o main.c main.o part3 write_to_shm.c write_to_shm.o
lee@lee-14ZD970-GX30K:~/homework/part2$
```

## ▶ Part 3

### 조건

50000이하의 두 자연수를 입력 받아 두 수의 최대 공약수와 최소 공배수를 출력하는 프로그램을 구현하시오

### 입력 25와 입력 45가 들어갔을 때

```
lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework/part2/part3$ ./Part3.out
50000자리 이하의 1번째 수를 입력하세요
25
50000자리 이하의 2번째 수를 입력하세요
45
최대 공약수는 5 입니다
최소 공배수는 225 입니다
lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework/part2/part3$
```

### 입력 23414와 입력 10232가 들어갔을 때

```
lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework/part2/part3$ ./Part3.out
50000자리 이하의 1번째 수를 입력하세요
23414
50000자리 이하의 2번째 수를 입력하세요
10232
최대 공약수는 2 입니다
최소 공배수는 119786024 입니다
lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework/part2/part3$
```

두가지 경우 모두 성공적으로 최대 공약수, 최소 공배수를 정확하게 계산한다.

#### [소스코드 관계도]

```
part2/part3:
Makefile common_divisor.c common_multiple.c find_min.c main.c my_numeric.h
```

Part3 는 main함수를 포함하는 main.c 와 문제에서 요구하는 최소공배수 알고리즘, 최대 공약수 알고리즘, 그리고 부가적으로 배열 중 그 값중 최소의 값을 찾는 코드가 각각 common\_multiple.c, common\_divisor.c, find\_min.c 에 작성되어있다. common\_multiple.c 코드는 common\_divisor함수를 포함하고, main.c 함수도 common\_divisor함수를 포함시킨다.

main.c	사용자에게 2 개의 정수값을 입력받아 배열에 저장한다. 이 저장된 값을 인자로, 함수 common_multiple, common_divisor를 호출하여, 최대공약수 최소공배수를 화면에 출력한다.
common_multiple.c,	2개의 수가 입력된다면, 최소 공배수는 두 수를 최대공약수로 나눈다. 이때 서로 소가 될 때까지 나누게 되며, 최소공배수는 최대공약수 * 서로소인 두 수의 곱이 최소공배수가 된다.
common_divisor.c,	사용자에게 입력받은 값 중 최소값 을 find_min 함수를 통하여 i 라고 설정하며, 이 설정된 i와 사용자의 값을 나머지 계산을 한다. 이때 i와 사용자의 값들이 모두 나머지가 0이면, 최대 공약수가 된다.
find_min.c	배열을 입력으로 받으며, 이 배열의 값 중 최소값을 반환한다.

#### [소스코드 분석]

```
for(int i = 0 ; i < INPUT_NUMBER ; ){
    printf(TO_STRING(_MAX_LIMIT_) "자리 이하의 %d번째 수를 입력하세요\n",i+1); // 50000자리 이하 몇번째 수 입력
    scanf("%d",&val[i]); // scanf로 입력받는다.
    if(val[i] < _MAX_LIMIT_+1) //만일 50000이하라면 다음값 입력
        i++;
    else
        printf(TO_STRING(_MAX_LIMIT_) "이하의 수를 입력해주세요\n"); //50000이상이면 재입력
}
```

사용자로부터 입력을 받는다. INPUT\_NUMBER는 매크로로 2개를 입력받는 문제이므로, 2로 설정이 되어있다.

```
g_common_divisor(val,sizeof(val)/sizeof(int));
l_common_multiple(val,sizeof(val)/sizeof(int));
```

함수 호출을 통하여 최대 공약수, 최소 공배수를 구한다.

```
for(int i = find_min(val,len) ; i > 0 ; i--){
    if(i == 1) return 1; //만일 a,b,c,d,e ... 의 수중 최소값이 1이면 최대공약수도 1이다.
    for(int j =0, k =0 ; j < len; j++){ //입력된 값들의 배열을 탐색한다.
        if( val[j] % i == 0) //입력된 수의 최소값 i 가 입력된 모든 값들과 나누어 나머지가 0일때까지 반복
            k++;
        if(k == len)
            return i; // 즉 i가 모든 수와 나누어떨어지면 i가 최대공약수
    }
}
```

최대 공약수 찾는 알고리즘:

1. 사용자 입력 (a,b) 중 작은 값을 find\_min 함수를 통하여 찾는다. 이 값을 i라고 한다.
2. i의 값을 줄여나가면서, i가 a,b 모두 나누어 떨어질 때까지 반복한다.
3. 만일 a,b 모두 나누어 떨어지면, 최대 공약수가 된다.

```
if( com != 1 ) //만일 최대공약수가 존재하면
{
    int* divided_val =malloc(sizeof(int)*len);
    memcpy(divided_val, val, len* sizeof(int) );

    for(int i =0; i <len; i++)
        divided_val[i] /= com; //구하고자 하는 수를 최대공약수로 나눈다.

    return com * l_common_multiple(divided_val, len); //최소공배수는 = 최대공약수 * 서로소인 두수
```

두 수가 서로소가 아니면, 즉 최대 공약수가 존재하면, 새로운 배열 divided\_val을 생성하고 이 배열을 최대 공약수로 나눈다. 이후 최대 공약수 \* (최대공약수로 나눈 배열)의 최소공배수를 구하면 된다. 이 과정은 두 수가 서로소일 때까지 반복이 되며, 마지막에는 아래와 같이 된다.

```
else//최대 공약수가 존재하지 않으면
{
    int mul_val=1;
    for(int i = 0; i < len ; i++)
        mul_val *= val[i];

    return mul_val; //서로소이면 두 서로소의 곱을 반환한다.
} //end else
```

최대 공약수가 1이면, 즉 두 수가 서로소이면, 두 수를 곱한 값을 반환한다.

## [Makefile 설계]

```
SUFFIXES: .c .o
CC = gcc
CFLAGS = -g

OBJS = main.o find_min.o common_divisor.o common_multiple.o
SRCS = $(OBJS:.o=.c)
HEADERS = my_numeric.h

TARGET = Part3.out

all : $(TARGET)

$(TARGET) : $(OBJS) $(HEADERS)
    $(CC) -o $@ $^

.PHONY: clean

clean :
    rm -f $(OBJS) $(TARGET)
```

→ .SUFFIXES 는 명시적 확장자로서 .o 파일이 없더라도 자동적으로 .c를 통하여 .o를 컴파일 한다.

→ CC를 통하여 gcc 컴파일러를 설정하였다.

→ CFLAGS 에 -g옵션을 주어서 gdb로 디버깅을 할 수 있도록 옵션을 추가 하였다.

→ OBJS는 소스파일의 오브젝트 파일이름을 포함하는 매크로 변수로, main함수가 존재하는 main.o 그리고 보조적인 함수 find\_min.o, common\_divisor.o, common\_multiple.o 등이 포함 되어있다.

→ TARGET = Part3.out 실행파일의 이름을 Part3.out으로 설정한다.

→ all: Makefile이 실행하여야 할 action으로, TARGET이 선행되어야 한다. TARGET은 아래에

**\$(TARGET):** 을 통해서, 필요한 선행조건을 찾을 수 있으며, 이때 \$(OBJS)를 필요로 하므로, OBJS를 찾게 된다. 명시적 .SUFFIXES 매크로 변수에 의하여 main.c ,find\_min.c, common\_divisor.c, common\_multiple.c 가 자동으로 main.o find\_min.o, common\_divisor.o, common\_multiple.o 로 컴파일 되어 타겟의 선행조건, \$(OBJS) 가 만족된다.

이후 \$(CC) -o \$(TARGET) \$(OBJS) 가 실행이 된다.

→ 위 표현은

find\_min.o, common\_divisor.o, common\_multiple.o  
my\_numeric.h 와 동일하며, 성공 시 Part3.out 실행이미지가 생성된다.

## make를 실행한 후

```
lee@lee-14ZD970-GX36K:~/homework/part2/part3$ make
gcc -g -c -o main.o main.c
gcc -g -c -o find_min.o find_min.c
gcc -g -c -o common_divisor.o common_divisor.c
gcc -g -c -o common_multiple.o common_multiple.c
gcc -o Part3.out main.o find_min.o common_divisor.o common_multiple.o my_numeric.h
lee@lee-14ZD970-GX36K:~/homework/part2/part3$ ls
Makefile  common_divisor.c  common_multiple.c  find_min.c  main.c  my_numeric.h
Part3.out  common_divisor.o  common_multiple.o  find_min.o  main.o
```



## ■ 문제 2

위 문제 1에서 구현한 c 파일들을 이용하여 (“복사하거나 옮기지 말고”), 상위 폴더인 homework1 폴더에 재귀적 make를 사용하는 최상위 Makefile을 구현하시오. 즉, 이 Makefile을 이용하여 make를 한번 수행하면 문제 1의 part1~3의 결과와 같이 object 및 실행 파일을 생성하여야 함. 또한 Makefile로부터 생성된 homework1 폴더에 생성된 실행 프로그램은 다음의 2가지를 수행하도록 하시오: 1) part1~3의 내용을 차례대로 수행하고 2) part2 프로그램의 두 출력(각 자릿수의 합과 곱)을 두 대각선의 길이로 갖는 마름모의 넓이를 출력하도록 구현하시오.

### 최상위 Makefile의 구현

```
SHELL := /bin/bash
CC= gcc
CFLAGS = -g
##TARGET LIST##
#####
TARGET1 = part1.out
TARGET2 = part2.out
TARGET3 = part3.out
TARGET4 = part4.out #문제 2번 파트 1,2,3 종합
OUT_SUFFIX = out
#####
OBJS = main.o read_from_shm.o func0.o

TARGET = $(TARGET1) $(TARGET2) $(TARGET3) $(TARGET4)
.PHONY : clean

all:    $(TARGET)
    @echo "Part1, Part2, Part3 프로그램 설치완료"

$(TARGET1):
    cd $* ; $(MAKE)

$(TARGET2):
    cd $* ; $(MAKE)

$(TARGET3):
    cd $(TARGET2:.$(OUT_SUFFIX)=)/$* ; $(MAKE)
$(TARGET4): $(OBJS)
    $(CC) -o $@ $^

clean:
    cd $(TARGET1:.$(OUT_SUFFIX)=); $(MAKE) clean
    cd $(TARGET2:.$(OUT_SUFFIX)=); $(MAKE) clean
    cd $(TARGET2:.$(OUT_SUFFIX=)/$(TARGET3:.$(OUT_SUFFIX)=); $(MAKE) clean
    rm -rf $(TARGET4) $(OBJS)
```

최상위 Makefile 은 총 4개의 Target이 존재한다.

TARGET1 = 문제 1번 Part1 의 메이크 파일을 실행시킨다.

TARGET2 = 문제 1번 Part2 의 메이크 파일을 실행시킨다.

TARGET3 = 문제 1번 Part3 의 메이크 파일을 실행시킨다.

TARGET4 = 문제 2번 최상위 메이크 파일을 실행시킨다.

all: TARGET, 즉 수행해야할 action은 모든 TARGET의 빌드이며, TARGET 은 TARGET = \$(TARGET1) \$(TARGET2) \$(TARGET3) \$(TARGET4) 의 집합이다.

즉 make는 \$(TARGET1)를 찾아, 레시피를 실행시키며, 그 레시피는 현재 디렉터리의 예하 Part1.out에서 .out을 제거한 확장자 (\$\* 매크로를 사용) 인 part1의 경로에 들어가며 메이크파일을 실행시킨다. 이 메이크파일은 앞서서 만든 part1폴더 내의 메이크파일이며, 이 메이크 파일이 실행되어 part1 내부에 실행파일 Part1.out 파일을 생성시킨다.

마찬가지로 make는 \$(TARGET2)를 찾아, 레시피를 실행시키며, 그 레시피는 현재 디렉터리의 예하 Part2.out에서 .out을 제거한 확장자 (\$\* 매크로를 사용) 인 part2의 경로에 들어가며 메이크파일을 실행시킨다. part2폴더 내의 메이크파일을 작동시키며, 이 메이크 파일이 실행되어 part2 내부에 실행파일 Part2.out 파일을 생성시킨다.

다음으로 make 는 \$(TARGET3)를 찾고 이 디렉터리는 part2 예하에 존재하기 때문에 part2/part3의 경로로 이동한다. 마찬가지로 make를 실행시켜, Part3.out을 실행시킨다.

문제2번 즉 part4.out은 \$(TARTGET4)를 통하여 만들어지며, 현재디렉토리에 OBJs를 선행조건으로 하여, Part4.out 실행파일을 생성시킨다.

제거의 과정은 앞에 과정과 동일하지만, make를 실행시키는 대신, make clean 과정을 실행시킨다. 이에 대한 레시피는 cd (디렉토리); make clean 과정을 거치면, 해당디렉토리 폴더에 마찬가지로 make clean이 작동하여 파일을 삭제한다.

#### 최상위 make 의 동작

```
Lee@Lee-14ZD970-GX30K:~/homework$ make
cd part1 ; make
make[1]: 디렉터리 '/home/lee/homework/part1' 들어감
gcc -g -c -o main.o main.c
gcc -g -c -o printarr.o printarr.c
gcc -g -c -o func1.o func1.c
gcc -o Part1.out main.o printarr.o func1.o
make[1]: 디렉터리 '/home/lee/homework/part1' 나감
cd part2 ; make
make[1]: 디렉터리 '/home/lee/homework/part2' 들어감
gcc -g -c -o main.o main.c
gcc -g -c -o func2.o func2.c
gcc -g -c -o write_to_shm.o write_to_shm.c
gcc -o Part2.out main.o func2.o write_to_shm.o extra.h
make[1]: 디렉터리 '/home/lee/homework/part2' 나감
cd part2/part3; make
make[1]: 디렉터리 '/home/lee/homework/part2/part3' 들어감
gcc -g -c -o main.o main.c
gcc -g -c -o find_min.o find_min.c
gcc -g -c -o common_divisor.o common_divisor.c
gcc -g -c -o common_multiple.o common_multiple.c
gcc -o Part3.out main.o find_min.o common_divisor.o common_multiple.o my_numeric.h
make[1]: 디렉터리 '/home/lee/homework/part2/part3' 나감
gcc -g -c -o main.o main.c
gcc -g -c -o read_from_shm.o read_from_shm.c
gcc -g -c -o func0.o func0.c
gcc -o part4.out main.o read_from_shm.o func0.o
Part1, Part2, Part3 프로그램 설치완료
Lee@Lee-14ZD970-GX30K:~/homework$
```

#### 최상위 make clean 의 동작

```

lee@lee-14ZD970-GX30K:~/homework$ make clean
cd part1; make clean
make[1]: 디렉터리 '/home/lee/homework/part1' 들어감
rm -f main.o printarr.o func1.o Part1.out
make[1]: 디렉터리 '/home/lee/homework/part1' 나감
cd part2; make clean
make[1]: 디렉터리 '/home/lee/homework/part2' 들어감
rm -f main.o func2.o write_to_shm.o Part2.out
make[1]: 디렉터리 '/home/lee/homework/part2' 나감
cd part2/part3; make clean
make[1]: 디렉터리 '/home/lee/homework/part2/part3' 들어감
rm -f main.o find_min.o common_divisor.o common_multiple.o Part3.out
make[1]: 디렉터리 '/home/lee/homework/part2/part3' 나감
rm -rf part4.out main.o read_from_shm.o func0.o
lee@lee-14ZD970-GX30K:~/homework$

```

### 조건

Makefile로부터 생성된 homework1 폴더에 생성된 실행 프로그램은 다음의 2가지를 수행하도록 하시오

- 1) part1~3의 내용을 차례대로 수행하고
- 2) part2 프로그램의 두 출력(각 자릿수의합과 곱)을 두 대각선의 길이로 갖는 마름모의 넓이를 출력하도록 구현하시오.

```

lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework$ ./part4.out
학번 입력: 201420703
214273

9자리의 수를 입력해주세요
123456789
합의 값: 45 곱의 값: 362880

50000자리 이하의 1번째 수를 입력하세요
24
50000자리 이하의 2번째 수를 입력하세요
36
최대 공약수는 12 입니다
최소 공배수는 72 입니다
마름모의 넓이는 8164800.000000입니다
lee@lee-14ZD970-GX30K:~/바탕화면/C_project/homework$

```

▶ 프로그램 part4.out을 실행시키면, part1, part2, part3의 프로그램이 연속하여 실행이 되며, part2에서의 합과 곱을 포함한 결과가 마름모의 넓이로 출력이 됨을 확인할 수 있다.

### [소스코드 관계도]

```

lee@lee-14ZD970-GX30K:~/homework$ ls . part1 part2 part2/part3
.:
Makefile func0.c main.c part1 part2 read_from_shm.c

```

Part4는 문제에서 요구하는 part1, part2, part3를 연속하여 실행하는 과정과 part2에서의 두 출력 (합과 곱)을 입력으로 하여, 마름모의 넓이를 구하는 문제를 해결해야 한다. 따라서 main함수를 포함하는 main.c와 마름모의 넓이를 구하는 func0.c, part2에서 실행되는 프로세서와 part4 프로그램의 ipc를 관리하는 read\_from\_shm.c 가 존재한다.

#### main.c

메인함수에는 part1.out, part2.out, part3.out의 실행경로가 저장되어있다. 프로그램은 fork()를 통하여 자식프로세스에 execv 함수를 통하여, 프로그램 1이미지와 다른 프로그램

	2, 3의 경로를 적재한다. 프로그램1이 종료시 마찬가지로 이 경로를 이용하여 프로그램2가 실행이 되며, 마찬가지로 프로그램3이 연속하여 실행이 된다. 자식프로세스가 종료가 되면, wait()를 통하여 기다린 부모프로세스가 마름모의 넓이를 구하는 과정을 진행한다. 이때 부모프로세스가 part2에서의 합과 곱을 전달받기위하여, 공유메모리를 활용하여, 이 데이터를 읽어와 출력한다. 이 함수는 read_from_shm.c에 정의되어있다.
read_from_shm.c,	동일한 KEY를 활용하여, 특정메모리에 8바이트의 데이터 즉 합과 곱을 읽어온다. part2에 write_to_shm.c 에는 반대로 특정메모리에 8바이트의 데이터를 기록하는 함수가 정의되어있다.
func0.c,	두 정수의 값을 입력받아 마름모의 넓이를 출력한다. 즉 (a,b) -> (double)a * b / 2 가 된다.

## [소스코드 분석]

```
int pid = fork();

if( pid < 0) return -1;

else if(pid == 0) //child
    execlp("./part1/Part1.out", "./part1/Part1.out", "./part2/Part2.out", "./part2/part3/Part3.out", NULL);
```

fork() 함수를 사용하여 자식프로세스를 만들고 이에

프로그램1 -> part1.out (이미지)

프로그램2 -> part2.out (경로)

프로그램3 -> part3.out (경로)

을 적재한다.

각 프로그램 part1, part2, part3에는 프로그램 종료직전 다음을 실행한다.

```
if(argc > 1) //만일 인자에 다음 프로그램 실행 파일의 위치가 존재하면
    execv(argv[1], &argv[1]); //종료하지 않고 다음 프로그램의 이미지를 덮어씌운다.
```

마지막 part3 프로그램은 argc = 1이므로, 더 이상 프로그램을 실행시키지 않고 반환된다.

마름모의 넓이를 구하기 위한, 합과 곱을 전달받기

```
int mem_id; //메모리 fd
void* mem_addr; // part4 프로그램과 공유할 메모리 주소
if ( -1 == ( mem_id = shmget( (key_t)KEY_NUM, MEM_SIZE, IPC_CREAT|0666))) exit(1);
if ( ( void *)-1 == ( mem_addr = shmat( mem_id, ( void *)0, 0))) exit(1);

((int*)mem_addr)[0] = sum; //합의 값 기록
((int*)mem_addr)[1] = mul; //곱의 값 기록
```

part2에서 공유메모리의 주소에 sum값과 mul값을 저장한다.

```
if ( -1 == ( mem_id = shmget( (key_t)KEY_NUM, MEM_SIZE, IPC_CREAT|0666))) return -1;
if ( ( void *)-1 == ( mem_addr = shmat( mem_id, ( void *)0, 0))) return -1;

(double)((int*)mem_addr)[0]*((int*)mem_addr)[1]/2)
```

part4에서 공유메모리의 주소에서 sum값과 mul값을 읽어온다.