*** 리눅스 관련 추가 명령어**

mv ../../third.c ./ third.c 파일을 현재 폴더로 이동
mv test.c class.c // test 파일을 class 파일로 변경
mv ../../homework ./ homework 폴더를 현재 폴더로 이동
mv sanghoonlee/*.c sanghoonlee/test/ 이상훈 폴더 모든 c 파일을 test 폴더로 옮김.

Cp -r test test2 디렉토리 복사

rm -rf *.c 현재 폴더 확장자 c 인 파일 지우기
rm -rf * 현재 폴더 모두 다 지우기

***오류 확인**

```
int main(void){
        int num=1;
        while(1){
                num+=num;
                if(num==100)
                        break;
        }
}
```

디버깅>

si - 호출 함수가 있다면 진입하면서 다음줄 확인.
ni - 호출 함수는 생략하고 다음줄 확인.
l – c 코드 전체확인 (list 의 줄임말).
p + 변수이름 - 변수의 값을 확인 가능하다.

-O0 옵션을 넣지 않으면 c 레벨 에서 코드단위 분석이 가능하다!

오류 의심 문구에 b 걸어두고 c 로 나머지 무시하고 다음 브레이크 까지 진행을 계속 가능 하다.

디버깅은 컴파일은 성공적임(즉 문법 오류는 없음)
하지만 논리적인 오류가 존재하는 경우에 수행하는 것 임. (위와 같은 오류는 2^n 으로 num 이 움직임)
그 외에도 예측치 못한 다양한 문제를 파악하기 위해서 디버깅을 하는 것이다.

프로그램이 동작은 하는데 이상한 동작을 보일 경우 하는 것이 디버깅이다.

B main → r → 프로그램 실행
기존에 si 활용 기계어를 분석했는데 c 레벨에서는 1 줄씩 이동이 가능하다 그럴경우에 s 나 n 을 입력한다.

c 언어에서 {} 는 scope 영역이라고 한다.
어셈블리 기반으로 해석을 하면
PUSH %rbp
MOV rsp rbp ㄱ
….. ㅣ 스코프 (스택 영역 생성) 이다.
RETQ ㅡ

{} 안에서만 지역변수가 생성되고 소멸한다는 것이다.

***static**
전역변수처럼 data 영역에 씌어진다.
차이점은 static 변수는 선언 된 함수 내에서만 변경이 가능하다.
메모리 동적 할당과 비슷한 개념으로 이해하면 된다. (but 해제 할 수 없다.)

***goto 의 이점과 CPU 파이프라인**

다중 반복문을 goto 없이 if 와 break 를 조합한 형태와 goto 로 처리하는 방법이 있다.
if 문은 기본적으로 mov, cmp, jmp 로 구성된다.
goto 는 jmp 하나로 끝이다.

for 문이 여러개 생기면 if, break 조합의 경우 for 문의 갯수만큼 m,c,j 가 생기게 된다.
문제는 바로 jmp 명령이다.

call 이나 jmp 를 CPU Instruction (명령어) 레벨에서 분기 명령어라고 하고 이들은
CPU 파이프라인에 매우 치명적인 손실을 가져다 준다.

기본적으로 아주 단순한 CPU 의 파이프라인을 설명하자면
아래와 같은 3 단계로 구성된다.

1. Fetch - 실행해야할 명령어을 물어옴
2. Decode - 어떤 명령어인지 해석함
3. Execute - 실제 명령어를 실행시킴.

파이프 라인이 짧은 것부터 긴 것이
5 단계 ~ 수십단계로 구성된다.
(ARM, INTEL 등등 다양한 프로세서들 모두 마찬가지)

그런데 왜 jmp 나 call 등의 분기 명령어가 문제가 될까?

기본적으로 분기 명령어는 파이프라인을 때려 부순다.
이뜻은 위의 가장 단순한 cpu 가 실행까지 3clock 을 소요하는데
파이프 라인이 깨지니
쓸때없이 또 다시 3clock 을 버려야 함을 의미한다.

만약 파이프라인의 단계가 수십 단계라면
분기가 여러 번 발생하면
파이프 라인 단계 * 분기 횟수만큼 CPU clock 을 낭비하게 된다.

즉 성능면에서도 goto 가 월등히 압도적이다.(jmp 1 번에 끝나니까)

ADD-MOV-CALL-MOV-MOV 의 순서로 진행 할 경우 (call 시 함수는 PUSH-MOV-SUB...)

Fetch-Decode-Execute (3 단계 파이프 라인인 경우)

ADD fetch	ADD decoding	ADD execute				
	MOV fetch	MOV decoding	MOV execute			
		CALL fetch	CALL decode	CALL excute		
			~~MOV~~	~~MOV~~	PUSH f	PUSH d
				~~MOV~~		MOV f

미리 실행이 되고 있었던 MOV 명령어가 폐기처분 되면서 낭비가 된다. (2 칸 딜레이 효과)

***SW 와 HW 동작을 생각 할 때 주의할 점**

SW 는 멀티코어 상황이 아니면 어떤 상황에서도 한번에 한가지 동작만 실행 할 수 있음

좀더 정확히 말하면 CPU 한개는 오로지 한 순간에 한 가지 동작만 할 수 있음.
반면 HW 회로는 병렬 회로가 존재하듯이 모든 회로가 동시에 동작 할 수 있다.
파이프 라인은 CPU 에 구성된 회로이기 때문에
모든 모듈들이 동시에 동작할 수 있는 것이다. (FPGA 프로그래밍은 병렬 동작임)

실제로 CPU 설계를 FPGA 가지고 함.

*** 재귀함수 호출**
(gbd) bt 는 함수 상태 확인 ?

n 함수를 거치지 않는 ; 단위 별로 호출
s 함수를 거치는 호출

scanf 까지 n 으로 호출하고 나서 다음 함수로 들어갈 때, bt 로 함수의 상태를 알아본다.
finish 는 지금까지 불러온 stack 함수들이 나온다.
그리고 현재 최종 함수의 return 값을 value 로 나온다. (다음 불러올 함수가 없는 경우에 finish..)
다음 함수의 상태를 알기 위해 다시 bt 를 사용한다.

* 피보나치 재귀함수 분석
0 1 1 2 3 5 8 13 ….

num=6 으로 두었을 때, 피보나치 재귀 함수를 분석하면,

main												
res												
fib(6)	-	-	-	-	-	-	-	|				
|	-	-	-	-	|			fib(4)	-	-	|	
fib(5)	-	-	|		fib(3)	-		|			fib(2)	
|			fib(2)		|	|		fib(3)			(1)	
fib(4)			(1)		fib(2)	fib(1)		|	-|			
|					(1)	(0)		fib(2)	fib(1)			
fib(3)	-							(1)	(0)			
|	|											
fib(2)	fib(1)											
(1)	(0)											

피보나치 재귀 함수를 분석하면 위 표와 같다.
왼쪽에 있는 함수 일수록 먼저 호출이 된다.
즉 Fib(6→5→4→3→2→1→2→3→2→1→4→3→2→1→2) 순으로 호출이 되며
각각의 리턴값이 더해져 5 가 나오게 된다.

* for 문 변환

1. 스키장에서 스키 장비를 임대하는데 37500 원이 든다.

또 3 일 이상 이용할 경우 20%를 할인 해준다.

일주일간 이용할 경우 임대 요금은 얼마일까 ?

(연산 과정은 모두 함수로 돌린다)

```
int Prob1(int days){
    int fee=37500;
    int sum=0;
    for(int i=1;i<=days;i++){
        if(i>3)
            sum+=(fee*0.8);
        else
            sum+=fee;
    }
    return sum;
}
```

3. 1 ~ 1000 사이에 3 의 배수의 합을 구하시오.

```
int Prob2(void){
    int sum=0;
    for(int i=1;i<=1000;i++){
        if(i%3==0)
            sum+=i;
    }
    return sum;
}
```

4. 1 ~ 1000 사이에 4 나 6 으로 나눠도 나머지가 1 인 수의 합을 출력하라.

```
int Prob3(void){
    int sum=0;
    for(int i=1;i<=1000;i++){
        if(i%4==1 && i%6==1)
            sum+=i;
    }
    return sum;
}
```

5. 7 의 배수로 이루어진 값들이 나열되어 있정한다.

함수의 인자(input)로 항의 갯수를 받아서 마지막 항의 값을 구하는 프로그램을 작성하라.

```
int Prob4(int size){
    if(size==1)
        return 7;
    else
        return Prob4(size-1)+7;
}
```

10. 구구단을 만들어보시오.

```
void Prob5(void){
    for(int i=1;i<10;i++){
        for(int j=1;j<10;j++){
            printf("%d * %d = %d\n",i,j,i*j);
        }
    }
}
```