

1번

```
#include<stdio.h>

int sale(int num1)
{
    num1=(num1*7)*0.8;
    return num1;
}

int main()
{
    int num1=37500,result;
    printf("할인 안된 일주일 스키 렌탈 가격 %d\n",num1*7);
    result=sale(num1);

    printf("할인 된 일주일 스키 렌탈 가격 %d\n",result);
    return 0;
}
```

3번

```
#include<stdio.h>
int main()
{
    int a=1,sum=0;
    while(a<=1000)
    {
        if(a%3==0)
        {
            sum=sum+a;
        }
        a++;
    }
    printf("1부터 1000사이 3의배수 합은 %d\n",sum);
    return 0;
}
~
```

4번

```
#include <stdio.h>

//first = start , second = end
void print_even(int start , int end)
{
    int i = start;
    while(i<=end)
    {
        if((i%24)==1)
        {
            printf("even = %d\n",i);
        }
        i++;
    }
}

int main(void)
{
    print_even(24,1000);
    return 0;
}
```

7번

```
(gdb) b*0x0000000000400546
Breakpoint 2 at 0x400546
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/minking/Homework/a.out

Breakpoint 2, 0x0000000000400546 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x0000000000400546 <+0>:      push    %rbp
    0x0000000000400547 <+1>:      mov     %rsp,%rbp
    0x000000000040054a <+4>:      sub     $0x10,%rsp
    0x000000000040054e <+8>:      mov     %fs:0x28,%rax
    0x0000000000400557 <+17>:     mov     %rax,-0x8(%rbp)
    0x000000000040055b <+21>:     xor     %eax,%eax
    0x000000000040055d <+23>:     mov     $0x0,%eax
    0x0000000000400562 <+28>:     mov     -0x8(%rbp),%rdx
    0x0000000000400566 <+32>:     xor     %fs:0x28,%rdx
    0x000000000040056f <+41>:     je      0x400576 <main+48>
    0x0000000000400571 <+43>:     callq  0x400420 <__stack_chk_fail@plt>
    0x0000000000400576 <+48>:     leaveq
    0x0000000000400577 <+49>:     retq
End of assembler dump.
(gdb)
```

shell clear

디버깅 화면 초기화.

r

프로그램 구동.

disas

기계어 보기.

<카페 작년 학습 자료?를 뒤져봐서 작성했습니다.>

?ip 레지스터 : 다음에 실행해야할 기계어의 주소를 나타냄

?ax 레지스터 : 주로 산술 연산에 활용하며

모든 return 값은 ax 레지스터에 저장됨

?cx 레지스터 : 반복을 하기 위해 사용하는 레지스터

?bp 레지스터 : Stack 의 베이스를 나타냄

?sp 레지스터 : Stack 의 최상위를 나타냄

- 앞에 이니셜이 r 이면 64 비트

e 면 32 비트

push %rbp

push 명령어 뒤에 있는 데이터를 밀어넣는다.

push 는 Stack 의 최상위에 값을 밀어넣는다.

push 는 sp 레지스터에 값을 밀어넣는다.

그리고 sp 레지스터는 포인터 크기만큼 증가한다.

[포인터의 크기 - 64 비트의 경우 8 byte - 32 비트의 경우 4 byte]

p/x \$레지스터이름

p/x \$rsp

sp ffdd38

bp 0x400570

어셈블리어 단위로 1 줄 진행하는 명령어는 si

C 언어 단위로 1 줄 진행하는 명령어는 s

현재 sp 의 메모리 주소에 bp 값이 있으므로

메모리 안을 들여다봐야 한다.

이때 사용하는 명령어는 x 다.

x \$rsp

sp ffdd30

mov 는 복사 명령어임

mov %rsp, %rbp 는 좌측의 레지스터 정보를

우측 레지스터로 복사한다.

p \$rbp

sub 는 subtract 의 약자로 뺄셈임

sub \$0x10, %rsp 는 rsp 에서 0x10 을 빼겠다는 뜻

sp ffdd20

'%' 는 레지스터 사용시 활용

'\$' 는 상수값 사용시 활용

call : 함수 호출

call = push + jmp

여기서 push 는 복귀 주소를 스택에 저장함

jmp 는 call 뒤에 오는 주소값으로 이동시킴

sp ffdd20

add 는 Addition 으로 덧셈임

add %eax, %eax 는 eax 끼리 더하고 결과를 eax 에 저장함

pop 은 Stack 의 최상위에서 값을 빼냄

이 빼낸값을 pop 뒤에 오는 녀석에게 전달함

ret 는 pop ip 와 동일한 기능을 수행함

10번 while 구구단

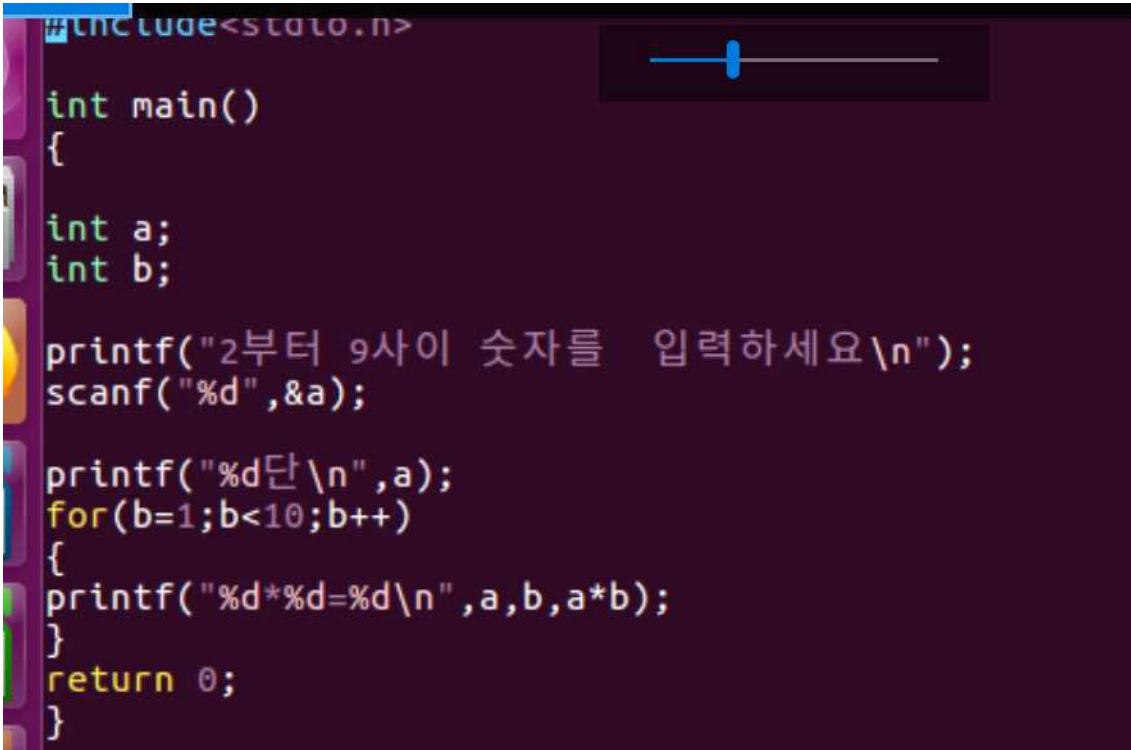
```
#include <stdio.h>

int main()
{
    int a=2;
    int b;

    while(a<10)
    {
        b=1;
        while(b<10)
        {
            printf("%dx%d=%d\n",a,b,a*b);
            b++;
        }
        a++;
        printf("\n");
    }

    return 0;
}
```

10번 scanf 구구단



```
#include <stdio.h>

int main()
{
    int a;
    int b;

    printf("2부터 9사이 숫자를 입력하세요\n");
    scanf("%d",&a);

    printf("%d단\n",a);
    for(b=1;b<10;b++)
    {
        printf("%d*%d=%d\n",a,b,a*b);
    }
    return 0;
}
```

12번 디버그

버그(bug)는 벌레를 뜻하며, 디버그(debug)는 원래 '해충을 잡다'라는 뜻이며, 프로그램의 오류를 벌레에 비유하여 오류를 찾아 수정하는 일이라는 의미로 쓰인다. 프로그램 개발공정의 마지막 단계에서 이루어진다. 주로 디버그가 오류수정 프로그램과 그 작업을 통칭하는 반면 작업에 중점을 둔 어휘는 디버깅(debugging)을 쓰며, 오류수정 소프트웨어를 가리킬 때는 디버거(debugger)라는 말을 쓴다.

디버그를 사용하는 이유?

프로그램을 개발하다가 에러가 발생하면 발생 위치 및 발생이유를 쉽게 찾을 수 있도록 도와준다. 컴파일에서 오류는 없지만 논리적 오류가 발생했을 때, 즉 실행은 잘 되지만 프로그램이 원하는 값이 나오지 않는 경우에 단계별로 프로그램을 실행시켜 가면서 실시간으로 프로그램의 상태를 볼 수 있으며 이로 인해 쉽게 오류를 찾아 낼 수 있다.

1. gcc -g -o 바꿀이름 본래이름.c

gdb를 사용하기 위해서 디버그를 할 수 있도록 설정.

2. gdb 디버깅실행할파일

디버깅 모드로 진입.

3. run (r)

프로그램 실행. breakpoint 가 설정되어 있을 시 breakpoint 까지만 실행.

4. break (b main)

main 라인에 breakpoint 설정합니다.

5. si

한줄 라인만큼 실행.

6. quit (q)

gdb 종료.

7. c

브레이크 포인트를 만날 때 까지 계속 진행.

8. bt

프로그램이 비정상적으로 종료 되었다면..

스택을 보기 위해 bt(backtrace) 명령어를 사용한다.

현재 활성화된 서브루틴의 스택 프레임을 출력해준다.

★ gbd 명령어

명령어	단축키	설명
break [행]	b	breakpoint 설정
break [함수명]		
delete [num]	d	breakpoint와 watchpoint 삭제
enable [num]	en	breakpoint와 watchpoint 활성화
disable [num]	dis	breakpoint와 watchpoint 비활성화
info breakpoints	info b	breakpoint와 watchpoint 정보
list	l	소스코드 출력
list [행], [행]		소스코드 범위만큼만 출력
run	r	디버깅 실행
step	s	현재 행 수행 후 정지, 함수 내부로 들어감
next	n	현재 행 수행 후 정지, 함수 호출 시 함수 수행 후 다음 행으로 감
continue	c	breakpoint를 만날 때까지 계속 진행
until	u	현재 루프를 나감
set listsize [num]	set li	list 출력라인 설정
print [변수명]	p	변수값 출력
display [변수명]	d	변수값 출력 유지
undisplay [num]	und	변수값 출력 해제
info display	info d	변수값 정보
info local	info lo	지역변수 출력
watch [변수명]	w	변수값이 변할 때 이전값과 현재값 출력

참고

<http://sosai.kr/128>

<http://kthan.tistory.com/entry/Linux%EB%A6%AC%EB%88%85%EC%8A%A4-%EB%94%94%EB%B2%84%EA%B9%85%EC%9D%84-%EC%9C%84%ED%95%9C-gdb-%EC%82%AC%EC%9A%A9%EB%B2%95-%EB%B0%8F-%EB%AA%85%EB%A0%B9%EC%96%B4-%EC%B4%88%EA%B8%89>

<http://bboy6604.tistory.com/entry/GDB%EB%A5%BC-%EC%82%AC%EC%9A%A9%ED%95%9C-Debug>

<http://air802.tistory.com/122>