

# Homework3

김민주

# 01



스키장에서 스키 장비를 임대하는데 37500원이 든다. 또 3일 이상 이용할 경우 20%를 할인 해준다.  
일주일간 이용할 경우 임대 요금은 얼마일까? (연산 과정은 모두 함수로 돌린다)

```
alswnqodrl@alswnqodrl-900X3K: ~/Homework
#include <stdio.h>
int payment(int days, int cost)
{
    if(days>=3)
        return cost*days*4/5;
    else
        return cost*days;
}

int main(void)
{
    int cost = 37500;
    int days = 7;
    printf("임대요금 %d\n", payment(days, cost));
    return 0;
}
```

# 01



스키장에서 스키 장비를 임대하는데 37500원이 든다. 또 3일 이상 이용할 경우 20%를 할인 해준다.  
일주일간 이용할 경우 임대 요금은 얼마일까 ? (연산 과정은 모두 함수로 돌린다)

```
alswnqodrl@alswnqodrl-900X3K:~/Homework$ vi h1.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ gcc h1.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ ./a.out
임대요금 210000
alswnqodrl@alswnqodrl-900X3K:~/Homework$ vi h1.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ gcc h1.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ ./a.out
임대요금 210000
alswnqodrl@alswnqodrl-900X3K:~/Homework$
```

# 02



1 ~ 1000사이에 3의 배수의 합을 구하시오.

```
alswnqodrl@alswnqodrl-900X3K: ~/Homework
#include <stdio.h>

int h3()
{
    int n=1;
    int sum=0;
    while(n<=1000)
    {
        if(!(n%3))
        {
            sum+=n;
        }
        n++;
    }
    return sum;
}

int main(void)
{
    printf("1~1000까지의 3의 배수 합 %d\n", h3());
    return 0;
}
~
~
```

# 02



1 ~ 1000사이에 3의 배수의 합을 구하시오.

```
alswnqodrl@alswnqodrl-900X3K: ~/Homework
alswnqodrl@alswnqodrl-900X3K:~$ cd Homework
alswnqodrl@alswnqodrl-900X3K:~/Homework$ vi h3.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ gcc h3.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ ./a.out
1~1000까지의 3의 배수 합 166833
alswnqodrl@alswnqodrl-900X3K:~/Homework$ vi h3.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$
```

# 03



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

```
alswnqodrl@alswnqodrl-900X3K: ~/Homework
#include <stdio.h>

int h4()
{
    int n=1;
    int sum=0;
    while(n<=1000)
    {
        if(n%4==1 || n%6==1)
        {
            sum+=n;
        }
        n++;
    }
    return sum;
}

int main(void)
{
    printf("\n 1~1000 중 4나 6으로 나뉘었을 때 나머지가 1인 수의 합 %d\n", h4());
    return 0;
}
~
~
~
```

# 03



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

```
alswnqodrl@alswnqodrl-900X3K: ~/Homework
^
alswnqodrl@alswnqodrl-900X3K:~/Homework$ vi h4.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ gcc h4.c
h4.c:14:1: error: expected identifier or '(' before 'return'
return sum;
^
h4.c:15:1: error: expected identifier or '(' before '}' token
}
^
alswnqodrl@alswnqodrl-900X3K:~/Homework$ vi h4.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ gcc h4.c
h4.c:14:1: error: expected identifier or '(' before 'return'
return sum;
^
h4.c:15:1: error: expected identifier or '(' before '}' token
}
^
alswnqodrl@alswnqodrl-900X3K:~/Homework$ vi h4.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ gcc h4.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ ./a.out

1~1000 중 4나 6으로 나뉘을 때 나머지가 1인 수의 합 166167
alswnqodrl@alswnqodrl-900X3K:~/Homework$ vi h4.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$
```

# 04



7의 배수로 이루어진 값들이 나열되어 있다고 가정한다.

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

```
alswnqodrl@alswnqodrl-900X3K: ~/Homewo
#include<stdio.h>

int h5(int i)
{
    return i*7;
}

int main(void)
{
    int n=0;
    printf("7의 배수의 n번째 항");
    scanf("%d",&n);
    printf("%d의 항은 %d \n",n,h5(n));
    return 0;
}
~
~
~
~
```



# 04



7의 배수로 이루어진 값들이 나열되어 있다고 가정한다.

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

```
alswnqodrl@alswnqodrl-900X3K: ~/Homework
h4.c:15:1: error: expected identifier or '(' before '}' token
}
^
alswnqodrl@alswnqodrl-900X3K:~/Homework$ vi h4.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ gcc h4.c
h4.c:14:1: error: expected identifier or '(' before 'return'
return sum;
^
h4.c:15:1: error: expected identifier or '(' before '}' token
}
^
alswnqodrl@alswnqodrl-900X3K:~/Homework$ vi h4.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ gcc h4.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ ./a.out

1~1000 중 4나 6으로 나뉘었을 때 나머지가 1인 수의 합 166167
alswnqodrl@alswnqodrl-900X3K:~/Homework$ vi h4.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ vi h5.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ gcc h5.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ ./a.out
7의 배수의 n번째 항5
5의 항은 35
```

# 05



C로 함수를 만들 때, **Stack**이란 구조가 생성된다.

이 구조가 어떻게 동작하는지 **Assembly Language**를 해석하며 기술해보시오.  
**esp, ebp, eip**등의 **Register**에 어떤 값이 어떻게 들어가는지 등등  
메모리에 어떤 값들이 들어가는지 등을 자세히 기술하시오.

```
alswnqodrl@alswnqodrl-900X3K: ~/
#include <stdio.h>
int mult2(int num)
{
    return num * 2;
}

int main(void)
{
    int i, sum = 0, result;
    for(i = 0; i < 5; i++)
        sum += i;
    result = mult2(sum);
    "h6.c" 26L, 207C
```

# 05



C로 함수를 만들 때, Stack이란 구조가 생성된다.

이 구조가 어떻게 동작하는지 Assembly Language를 해석하며 기술해보시오.

esp, ebp, eip등의 Register에 어떤 값이 어떻게 들어가는지 등등

메모리에 어떤 값들이 들어가는지 등을 자세히 기술하시오.

```
(gdb) disas
Dump of assembler code for function main:
0x00000000004004e4 <+0>:      push    %rbp
0x00000000004004e5 <+1>:      mov     %rsp,%rbp
0x00000000004004e8 <+4>:      sub     $0x10,%rsp
=> 0x00000000004004ec <+8>:      movl    $0x0,-0x8(%rbp)
0x00000000004004f3 <+15>:     movl    $0x0,-0xc(%rbp)
0x00000000004004fa <+22>:     jmp     0x400506 <main+34>
0x00000000004004fc <+24>:     mov     -0xc(%rbp),%eax
0x00000000004004ff <+27>:     add     %eax,-0x8(%rbp)
0x0000000000400502 <+30>:     addl    $0x1,-0xc(%rbp)
0x0000000000400506 <+34>:     cmpl    $0x4,-0xc(%rbp)
0x000000000040050a <+38>:     jle     0x4004fc <main+24>
0x000000000040050c <+40>:     mov     -0x8(%rbp),%eax
0x000000000040050f <+43>:     mov     %eax,%edi
0x0000000000400511 <+45>:     callq   0x4004d6 <mult2>
0x0000000000400516 <+50>:     mov     %eax,-0x4(%rbp)
0x0000000000400519 <+53>:     mov     $0x0,%eax
0x000000000040051e <+58>:     leaveq  0
0x000000000040051f <+59>:     retq
End of assembler dump.
(gdb) █
```

Push %rsp : 현재 %rsp에 %rbp를 쌓는다.

Mov %rsp, %rbp: 현재 rsp값을 rbp로 이동시킨다.

Sub \$0x10, %rsp: 16byte를 rsp에서 빼주어 stack으로 사용한다.

movl \$0x0, -0x8(%rbp): 0byte를 rbp기준 8byte하위로 이동한다.

movl \$0x0, -0xc(%rbp): 0byte를 rbp기준 12byte하위로 이동한다.

Jmp 0x400506 <main+34>: 0x400506 주소로 jump하여  
main+34을 실행한다.(반복문 시작)

mov -0xc(%rbp), %eax: rbp기준 하위 12byte를 eax로 이동한다.  
(register로 값 저장)

add %eax, -0x8(%rbp) : %eax 에 저장된 값을 rbp기준 하위  
8byte에 저장된 값과 더한다.

Cmpl \$0x4, -0xc(%rbp): 4byte와 rbp 하의 12byte에 저장된 정보를

# 06



구구단을 만들어보시오.

```
alswnqodrl@alswnqodrl-900X3K: ~/Homework
#include<stdio.h>

int h7()
{
    int i=1,j=1;
    while(i<10)
    {
        printf("\n%d단\n",i);
        j=1;
        while(j<10)
        {
            printf("%d*%d = %d\n",i,j,i*j);
            j++;
        }
        i++;
    }
}

int main(void)
{
    printf("구구단\n");
    h7();
    return 0;
}
~
~
```

# 06

구구단을 만들어보시오.

```
alswnqodrl@alswnqodrl-900X3K: ~/Homework
alswnqodrl@alswnqodrl-900X3K:~/Homework$ vi h7.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ gcc h7.c
alswnqodrl@alswnqodrl-900X3K:~/Homework$ ./a.out
구구단

1단
1*1 = 1
1*2 = 2
1*3 = 3
1*4 = 4
1*5 = 5
1*6 = 6
1*7 = 7
1*8 = 8
1*9 = 9

2단
2*1 = 2
2*2 = 4
2*3 = 6
2*4 = 8
2*5 = 10
2*6 = 12
2*7 = 14
2*8 = 16
2*9 = 18

3단
3*1 = 3
3*2 = 6
3*3 = 9
3*4 = 12
3*5 = 15
3*6 = 18
3*7 = 21
3*8 = 24
3*9 = 27

4단
4*1 = 4
4*2 = 8
4*3 = 12
4*4 = 16
4*5 = 20
4*6 = 24
4*7 = 28
4*8 = 32
4*9 = 36
```

```
5단
5*1 = 5
5*2 = 10
5*3 = 15
5*4 = 20
5*5 = 25
5*6 = 30
5*7 = 35
5*8 = 40
5*9 = 45

6단
6*1 = 6
6*2 = 12
6*3 = 18
6*4 = 24
6*5 = 30
6*6 = 36
6*7 = 42
6*8 = 48
6*9 = 54

7단
7*1 = 7
7*2 = 14
7*3 = 21
7*4 = 28
7*5 = 35
7*6 = 42
7*7 = 49
7*8 = 56
7*9 = 63

8단
8*1 = 8
8*2 = 16
8*3 = 24
8*4 = 32
8*5 = 40
8*6 = 48
8*7 = 56
8*8 = 64
8*9 = 72

9단
9*1 = 9
9*2 = 18
9*3 = 27
9*4 = 36
9*5 = 45
9*6 = 54
9*7 = 63
9*8 = 72
9*9 = 81
alswnqodrl@alswnqodrl-900X3K:~/Homework$
```

# 07



Visual Studio에서 Debugging하는 방법에 대해 기술해보시오.

Break Point는 어떻게 잡으며, 조사식, 메모리, 레지스터등의 디버그 창은

각각 어떤 역할을 하고 무엇을 알고자 할 때 유용한지 기술하시오.

```
sdrl@alswnqodrl-Z20NH-AS51B1U: ~/my_proj/Homework/sanghoonlee
0x000000000040055b <+38>: mov    $0x0,%eax
0x0000000000400560 <+43>: callq 0x400400 <printf@plt>
0x0000000000400565 <+48>: mov    $0x0,%eax
0x000000000040056a <+53>: leaveq
0x000000000040056b <+54>: retq
End of assembler dump.
(gdb) p/x $rbp
$25 = 0x7fffffffdc50
(gdb) si
13      res = myfunc(num);
(gdb) disas
Dump of assembler code for function main:
0x0000000000400535 <+0>: push    %rbp
0x0000000000400536 <+1>: mov     %rsp,%rbp
0x0000000000400539 <+4>: sub     $0x10,%rsp
0x000000000040053d <+8>: movl    $0x3,-0x8(%rbp)
=> 0x0000000000400544 <+15>: mov     -0x8(%rbp),%eax
0x0000000000400547 <+18>: mov     %eax,%edi
0x0000000000400549 <+20>: callq   0x400526 <myfunc>
0x000000000040054e <+25>: mov     %eax,-0x4(%rbp)
0x0000000000400551 <+28>: mov     -0x4(%rbp),%eax
0x0000000000400554 <+31>: mov     %eax,%esi
0x0000000000400556 <+33>: mov     $0x4005f4,%edi
0x000000000040055b <+38>: mov     $0x0,%eax
0x0000000000400560 <+43>: callq   0x400400 <printf@plt>
0x0000000000400565 <+48>: mov     $0x0,%eax
0x000000000040056a <+53>: leaveq
0x000000000040056b <+54>: retq
End of assembler dump.
(gdb) p/x $rbp
$26 = 0x7fffffffdc50
(gdb) si
```

Push %rsp : 현재 %rsp에 %rbp를 쌓는다.

Mov %rsp, %rbp: 현재 rsp값을 rbp로 이동시킨다.

Sub \$0x10, %rsp: 16byte를 rsp에서 빼주어 stack으로 사용한다.

movl \$0x3, -0x8(%rbp): 3byte를 rbp기준 8byte하위로 이동한다.

mov -0x8(%rbp), %eax: rbp기준 하위 8byte를 eax로 이동한다.  
(register로 값 저장)

mov %eax, %edi: %eax 에 저장된 값을 edi로 이동한다.

Callq 0x480526 <myfunc> ; 0x480526 주소에서 myfunc함수를  
실행한 후 빠져  
여나와 다음 주소로 복귀한다.

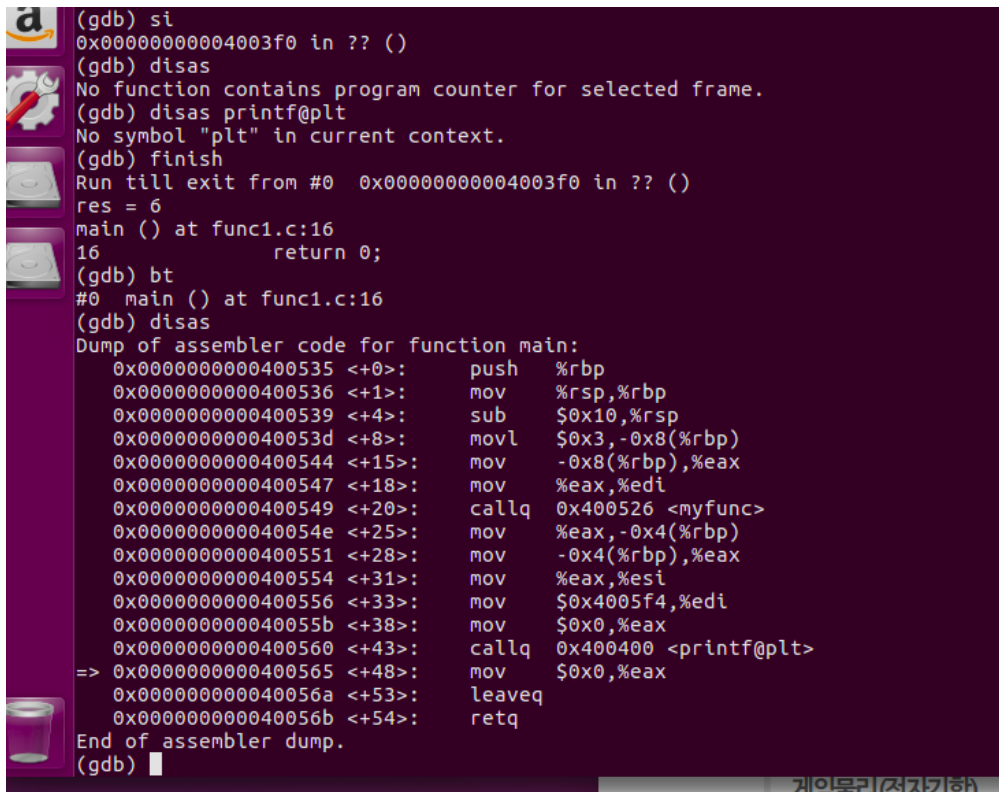
# 07



Visual Studio에서 Debugging하는 방법에 대해 기술해보시오.

Break Point는 어떻게 잡으며, 조사식, 메모리, 레지스터등의 디버그 창은

각각 어떤 역할을 하고 무엇을 알고자 할 때 유용한지 기술하시오.



```
(gdb) si
0x000000004003f0 in ?? ()
(gdb) disas
No function contains program counter for selected frame.
(gdb) disas printf@plt
No symbol "plt" in current context.
(gdb) finish
Run till exit from #0 0x000000004003f0 in ?? ()
res = 6
main () at func1.c:16
16      return 0;
(gdb) bt
#0  main () at func1.c:16
(gdb) disas
Dump of assembler code for function main:
0x00000000400535 <+0>:  push    %rbp
0x00000000400536 <+1>:  mov     %rsp,%rbp
0x00000000400539 <+4>:  sub     $0x10,%rsp
0x0000000040053d <+8>:  movl    $0x3,-0x8(%rbp)
0x00000000400544 <+15>:  mov     -0x8(%rbp),%eax
0x00000000400547 <+18>:  mov     %eax,%edi
0x00000000400549 <+20>:  callq   0x400526 <myfunc>
0x0000000040054e <+25>:  mov     %eax,-0x4(%rbp)
0x00000000400551 <+28>:  mov     -0x4(%rbp),%eax
0x00000000400554 <+31>:  mov     %eax,%esi
0x00000000400556 <+33>:  mov     $0x4005f4,%edi
0x0000000040055b <+38>:  mov     $0x0,%eax
0x00000000400560 <+43>:  callq   0x400400 <printf@plt>
=> 0x00000000400565 <+48>:  mov     $0x0,%eax
0x0000000040056a <+53>:  leaveq  0
0x0000000040056b <+54>:  retq
End of assembler dump.
(gdb)
```

Call 문을 빠져나온 후 모든 과정이 거의 동일함

다만 두 번째 call문을 만났을 때  
값이 0로 향해 빠져나오지 못하는 상황이 발생하는데  
이 때 bt를 사용하면 call문 다음의 주소로 넘어가는 것이  
가능해진다.

bt는 오류가 발생한 함수를 역으로 찾아갈 수 있도록  
해준다.

한편 c를 통해 c: 다음 브레이크 포인트를 만날 때까지  
계속해서 시행하게 할 수도 있다.