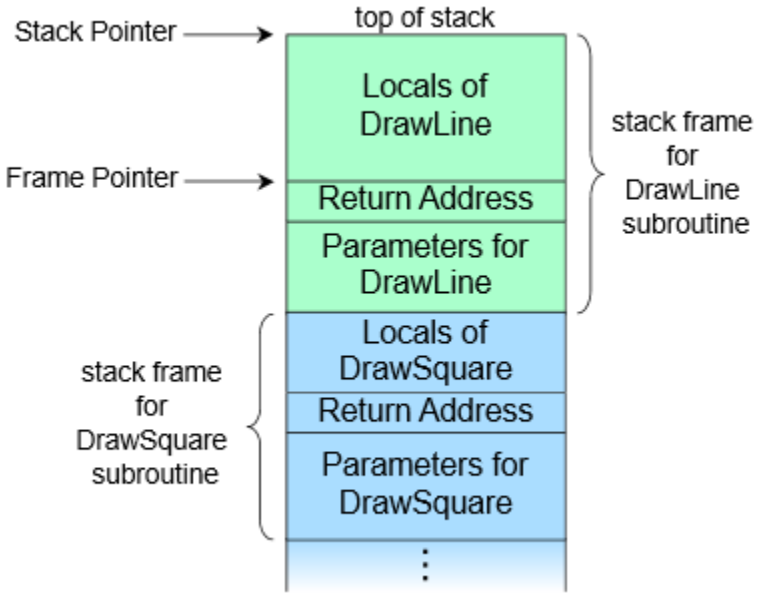
C언어 - HW5

임베디드스쿨2기
Lv1과정
2021. 04. 12
차현호

Call stack

Call stack이란 서브루틴(함수) 호출시 다음에 실행될 주소와 rbp같은 레지스터 값을 스택에 저장해두어 서브루틴 반환시 원래 실행하던 구문으로 돌아올 수 있게 해준다.

Call stack은 우리가 이제껏 봐왔던 함수호출시에 실행되는 push %rbp과 관련이 있다.



출처 : 구글 위키

Switch문 분석

아래 c 코드는 switch문을 사용한 c코드이다. 어셈블리어를 보며 분석한다.

```c
#include <stdio.h>

// #define을 통해서 특정 숫자를 아래와 같은 매크로
// 옵션들이 많아지면 숫자들이 많아지기 때문에
// 일일히 숫자로 기억하는 것 보다 옵션과 관련된 명

#define BLDC        1
#define SERVO       2
#define CAMERA      3
#define AUDIO       4

int main(void)
{
    int num;

    printf("숫자를 입력하세요: ");
    scanf("%d", &num);

    // switch 문의 경우
    // 소괄호 내부의 num을 판정하고
    // case에 해당하는 내용이 있다면 해당 내용을
    // 이때 break를 만나지 않는다면 멈추지 않는다.
    switch (num)
    {
        case BLDC:
            printf("BLDC 모터 동작\n");
            // break;

        case SERVO:
            printf("SERVO 모터 구동\n");
            break;

        case CAMERA:
            printf("카메라 디바이스 활성화\n");
            break;

        case AUDIO:
            printf("오디오 코덱 활성화\n");
            break;
    }

    return 0;
}
```

Switch문 분석

```
(gdb) info registers $rsp
rsp            0x7fffffffdf48    0x7fffffffdf48
```

시작할때의 rsp 레지스터의 값은 0x7fffffffdf48 이다
push %rbp -> 현재 rbp레지스터의 값을 스택에 넣는다. 그리고 rsp는 -8된다.
mov %rsp,%rbp -> rbp레지스터에 rsp값을 대입한다.
sub $0x10,%rsp -> rsp 레지스터 값에서 16바이트만큼 뺀다 (변수할당)
mov %fs:0x28,%rax -> 커널 관련
mov %rax,-0x8(%rbp) -> rax 레지스터 값을 rbp-8만큼 뺀 주소에 넣는다.
xor %eax,%eax -> 자기자신과 xor 연산 eax레지스터가 0으로 된다.
lea 0x120(%rip),%rdi -> rip레지스터의 주소값보다 0x120더한 주소를 rdi레지스터에대입한다.
mov $0x0,%eax -> eax레지스터의 값을 0으로 바꾼다.

```
0x0000555555554769a <+0>:     push   %rbp
0x0000555555554769b <+1>:     mov    %rsp,%rbp
0x0000555555554769e <+4>:     sub    $0x10,%rsp
0x000055555555476a2 <+8>:     mov    %fs:0x28,%rax
0x000055555555476ab <+17>:    mov    %rax,-0x8(%rbp)
0x000055555555476af <+21>:    xor    %eax,%eax
0x000055555555476b1 <+23>:    lea    0x120(%rip),%rdi
0x000055555555476b8 <+30>:    mov    $0x0,%eax
```

Switch문 분석

```
0x00005555555547aa <+64>:    mov    -0xc(%rbp),%eax
0x00005555555547ad <+67>:    cmp    $0x2,%eax
0x00005555555547b0 <+70>:    je     0x5555555547d8 <main+110>
0x00005555555547b2 <+72>:    cmp    $0x2,%eax
0x00005555555547b5 <+75>:    jg     0x5555555547be <main+84>
0x00005555555547b7 <+77>:    cmp    $0x1,%eax
0x00005555555547ba <+80>:    je     0x5555555547ca <main+96>
0x00005555555547bc <+82>:    jmp    0x555555554801 <main+151>
0x00005555555547be <+84>:    cmp    $0x3,%eax
0x00005555555547c1 <+87>:    je     0x5555555547e6 <main+124>
0x00005555555547c3 <+89>:    cmp    $0x4,%eax
0x00005555555547c6 <+92>:    je     0x5555555547f4 <main+138>
0x00005555555547c8 <+94>:    jmp    0x555555554801 <main+151>
0x00005555555547ca <+96>:    lea    0xf6(%rip),%rdi        # 0x5555555548c7
0x00005555555547d1 <+103>:   callq  0x555555554610 <puts@plt>
0x00005555555547d6 <+108>:   jmp    0x555555554801 <main+151>
0x00005555555547d8 <+110>:   lea    0xfb(%rip),%rdi        # 0x5555555548da
0x00005555555547df <+117>:   callq  0x555555554610 <puts@plt>
0x00005555555547e4 <+122>:   jmp    0x555555554801 <main+151>
0x00005555555547e6 <+124>:   lea    0x103(%rip),%rdi        # 0x5555555548f0
0x00005555555547ed <+131>:   callq  0x555555554610 <puts@plt>
0x00005555555547f2 <+136>:   jmp    0x555555554801 <main+151>
0x00005555555547f4 <+138>:   lea    0x116(%rip),%rdi        # 0x555555554911
0x00005555555547fb <+145>:   callq  0x555555554610 <puts@plt>
0x0000555555554800 <+150>:   nop
```

첨부되어있는 그림의
부분이 switch문 부분이다.

Switch문 분석

먼저 scanf함수로부터 입력받은 값(%rbp-0xc)을 eax레지스터에 대입한다.

그다음 cmp $0x2, %eax부분은 eax레지스터의 값(현재 1)과 2를 뺀다.
그리고 je점프문을 실행하는데 위의 cmp 결과가 0이면 점프하는 구문이다. 우리의경우 0이 아니므로 다음 구문으로 넘어간다.

아래의 어셈블리어부분에 해당하는 c코드도 아래에 첨부하였다.

```
0x00005555555547aa <+64>:    mov    -0xc(%rbp),%eax
0x00005555555547ad <+67>:    cmp    $0x2,%eax
0x00005555555547b0 <+70>:    je     0x5555555547d8 <main+110>
```

```
case SERVO:
    printf("SERVO 모터 구동\n");
    break;
```

Switch문 분석

그다음의 시작은 cmp $0x2,%eax인데 마찬가지로 우리가 입력한값(1)에서 2를 뺀다.
그후 jg 점프문을 실행하는데 jg 점프문은 cmp결과가 양수이면 점프하는 구문이다.

우리는 해당안되므로 다음구문으로 진행된다 그러나 만약 입력값이 3 또는 4이면 점프하여 또 비교구문을 만나게 된다. 이부분은 뒤에서 또 설명한다.

```
0x00005555555547b2 <+72>:    cmp    $0x2,%eax
0x00005555555547b5 <+75>:    jg     0x5555555547be <main+84>
```

Switch문 분석

다음구문으로 넘어가게 되면 또다시 비교구문이 나오는데 cmp $0x1,%eax 후에 je는 앞에서 살펴봤듯이 cmp값이 0이면 점프하게 되는 구문이다.

우리의 입력값1과 cmp $0x1 결과는 0이므로 je 구문이 실행되어 0x5555555547ca 주소로 점프하게 된다.

```
0x00005555555547b7 <+77>:    cmp    $0x1,%eax
0x00005555555547ba <+80>:    je     0x5555555547ca <main+96>
```

```
case BLDC:
    printf("BLDC 모터 동작\n");
    break;
```

Continue문 분석

아래의 c코드는 continue를 사용한 c코드이다. switch문과 마찬가지로 분석해본다.

```c
#include <stdio.h>

int main(void)
{
    int i, num;

    printf("1 ~ n까지 출력합니다. (n을 선택하세요): ");
    scanf("%d", &num);

    for (i = 1; i <= num; i++)
    {
        if (!(i % 3))
        {
            // 다시 위로 돌아감(증감부를 수행하게됨)
            // 결국 아래의 printf를 실행하지 않고 스킵하게 됨
            continue;
        }

        printf("i = %3d\n", i);
    }

    return 0;
}
```

Continue문 분석

어셈블리어를 살펴보면 printf문 전까지의 명령어는 이전의 switch문과 거의 유사하다.

```
0x000055555555471a <+0>:     push   %rbp
0x000055555555471b <+1>:     mov    %rsp,%rbp
0x000055555555471e <+4>:     sub    $0x10,%rsp
0x0000555555554722 <+8>:     mov    %fs:0x28,%rax
0x000055555555472b <+17>:    mov    %rax,-0x8(%rbp)
0x000055555555472f <+21>:    xor    %eax,%eax
0x0000555555554731 <+23>:    lea    0x120(%rip),%rdi
0x0000555555554738 <+30>:    mov    $0x0,%eax
```

Continue문 분석

아래의 movl $0x1, -0xc(%rbp)는 for문에서 (i=1)에 해당하는 부분이며 다음의 jmp는 for문의 조건을 비교하기위한 곳으로 점프한다.

```
0x000055555555475a <+64>:    movl   $0x1,-0xc(%rbp)
0x0000555555554761 <+71>:    jmp    0x5555555547a3 <main+137>
```

```
0x0000555555547a3 <+137>:    mov    -0x10(%rbp),%eax
0x0000555555547a6 <+140>:    cmp    %eax,-0xc(%rbp)
0x0000555555547a9 <+143>:    jle    0x555555554763 <main+73>
```

%rbp-0x10는 우리가 입력한 n값이 저장되어있는 메모리 주소이며, %rbp-0xc는 i값이 저장되어있는 메모리 주소이다. 이 두개의 값을 cmp로 비교한다. 이때 비교결과가 0이아니므로 0x555555554763으로 점프하지 않는다.

Continue문 분석

아래의 movl $0x1, -0xc(%rbp)는 for문에서 (i=1)에 해당하는 부분이며 다음의 jmp는 for문의 조건을 비교하기위한 곳으로 점프한다.

```
0x000055555555475a <+64>:    movl   $0x1,-0xc(%rbp)
0x0000555555554761 <+71>:    jmp    0x5555555547a3 <main+137>
```

```
0x0000555555547a3 <+137>:    mov    -0x10(%rbp),%eax
0x0000555555547a6 <+140>:    cmp    %eax,-0xc(%rbp)
0x0000555555547a9 <+143>:    jle    0x555555554763 <main+73>
```

%rbp-0x10는 우리가 입력한 n값이 저장되어있는 메모리 주소이며, %rbp-0xc는 i값이 저장되어있는 메모리 주소이다. 이 두개의 값을 cmp로 비교한다. 그다음 jle에의해 0x555555554763으로 점프한다.

Continue문 분석

Mmov –0xc(%rbp),%eax -> i값(1)을 ecx레지스터에 대입
Mmov $0x55555556,%edx -> 0x55555556을 edx레지스터에 대입
Mmov %ecx,%eax -> ecx레지스터값을 eax레지스터에 대입
Iimul %edx -> edx레지스터값 x 0을 해줌으로써 0으로 초기화
Mmov %ecx,%eax -> ecx레지스터값을 eax레지스터에 대입
Ssar $0x1f,%eax -> eax레지스터값을 부호를 유지하면서 32비트만큼 오른쪽 쉬프트한다 결과 eax는 0
<+98>까지는 전부 결과가 0이다.

```
0x0000555555554763 <+73>:    mov    -0xc(%rbp),%ecx
0x0000555555554766 <+76>:    mov    $0x55555556,%edx
0x000055555555476b <+81>:    mov    %ecx,%eax
0x000055555555476d <+83>:    imul   %edx
0x000055555555476f <+85>:    mov    %ecx,%eax
0x0000555555554771 <+87>:    sar    $0x1f,%eax
0x0000555555554774 <+90>:    sub    %eax,%edx
0x0000555555554776 <+92>:    mov    %edx,%eax
0x0000555555554778 <+94>:    mov    %eax,%edx
0x000055555555477a <+96>:    add    %edx,%edx
0x000055555555477c <+98>:    add    %eax,%edx
0x000055555555477e <+100>:   mov    %ecx,%eax
0x0000555555554780 <+102>:   sub    %edx,%eax
0x0000555555554782 <+104>:   test   %eax,%eax
0x0000555555554784 <+106>:   je     0x55555555479e <main+132>
```

Continue문 분석

mov –0xc(%rbp),%eax -> i값(1)을 ecx레지스터에 대입
mov $0x55555556,%edx -> 0x55555556을 edx레지스터에 대입
mov %ecx,%eax -> ecx레지스터값을 eax레지스터에 대입
imul %edx -> edx레지스터값 x 0을 해줌으로써 0으로 초기화
mov %ecx,%eax -> ecx레지스터값을 eax레지스터에 대입
sar $0x1f,%eax -> eax레지스터값을 부호를 유지하면서 32비트만큼 오른쪽 쉬프트한다 결과 eax는 0
<+98>까지는 전부 결과가 0이다.

```
0x0000555555554763 <+73>:    mov    -0xc(%rbp),%ecx
0x0000555555554766 <+76>:    mov    $0x55555556,%edx
0x000055555555476b <+81>:    mov    %ecx,%eax
0x000055555555476d <+83>:    imul   %edx
0x000055555555476f <+85>:    mov    %ecx,%eax
0x0000555555554771 <+87>:    sar    $0x1f,%eax
0x0000555555554774 <+90>:    sub    %eax,%edx
0x0000555555554776 <+92>:    mov    %edx,%eax
0x0000555555554778 <+94>:    mov    %eax,%edx
0x000055555555477a <+96>:    add    %edx,%edx
0x000055555555477c <+98>:    add    %eax,%edx
0x000055555555477e <+100>:   mov    %ecx,%eax
0x0000555555554780 <+102>:   sub    %edx,%eax
0x0000555555554782 <+104>:   test   %eax,%eax
0x0000555555554784 <+106>:   je     0x55555555479e <main+132>
```

Continue문 분석

mov %ecx,%eaxx -> ecx의값(i=1)을 eax레지스터에 대입한다.
test %eax,%eax -> 두개의 레지스터값을 AND연산하여 비교한다.
Je 0x55555555479e -> test의 결과가 equl이므로 점프한다.

```
0x0000555555554763 <+73>:    mov    -0xc(%rbp),%ecx
0x0000555555554766 <+76>:    mov    $0x55555556,%edx
0x000055555555476b <+81>:    mov    %ecx,%eax
0x000055555555476d <+83>:    imul   %edx
0x000055555555476f <+85>:    mov    %ecx,%eax
0x0000555555554771 <+87>:    sar    $0x1f,%eax
0x0000555555554774 <+90>:    sub    %eax,%edx
0x0000555555554776 <+92>:    mov    %edx,%eax
0x0000555555554778 <+94>:    mov    %eax,%edx
0x000055555555477a <+96>:    add    %edx,%edx
0x000055555555477c <+98>:    add    %eax,%edx
0x000055555555477e <+100>:   mov    %ecx,%eax
0x0000555555554780 <+102>:   sub    %edx,%eax
0x0000555555554782 <+104>:   test   %eax,%eax
0x0000555555554784 <+106>:   je     0x55555555479e <main+132>
```

Continue문 분석

mov %ecx,%eaxx -> ecx의값(i=1)을 eax레지스터에 대입한다.
test %eax,%eax -> 두개의 레지스터값을 AND연산하여 비교한다.
Je 0x55555555479e -> test의 결과가 1이고 0이아니므로 다음구문을 실행한다.

```
0x0000555555554763 <+73>:    mov    -0xc(%rbp),%ecx
0x0000555555554766 <+76>:    mov    $0x55555556,%edx
0x000055555555476b <+81>:    mov    %ecx,%eax
0x000055555555476d <+83>:    imul   %edx
0x000055555555476f <+85>:    mov    %ecx,%eax
0x0000555555554771 <+87>:    sar    $0x1f,%eax
0x0000555555554774 <+90>:    sub    %eax,%edx
0x0000555555554776 <+92>:    mov    %edx,%eax
0x0000555555554778 <+94>:    mov    %eax,%edx
0x000055555555477a <+96>:    add    %edx,%edx
0x000055555555477c <+98>:    add    %eax,%edx
0x000055555555477e <+100>:   mov    %ecx,%eax
0x0000555555554780 <+102>:   sub    %edx,%eax
0x0000555555554782 <+104>:   test   %eax,%eax
0x0000555555554784 <+106>:   je     0x55555555479e <main+132>
```

Continue문 분석

mov %ecx,%eax -> ecx의값(i=1)을 eax레지스터에 대입한다.
test %eax,%eax -> 두개의 레지스터값을 AND연산하여 비교한다.
je 0x55555555479e -> test의 결과가 1이고 0이아니므로 다음구문을 실행한다.
위의 test와 je부분이 c코드의 continue 부분이다.

만약 조건을 충적하여 je가 점프하게되면 printf함수를 건너뛰고 i + 1이 된채 for문이 실행된다.

```
0x0000555555554763 <+73>:    mov    -0xc(%rbp),%ecx
0x0000555555554766 <+76>:    mov    $0x55555556,%edx
0x000055555555476b <+81>:    mov    %ecx,%eax
0x000055555555476d <+83>:    imul   %edx
0x000055555555476f <+85>:    mov    %ecx,%eax
0x0000555555554771 <+87>:    sar    $0x1f,%eax
0x0000555555554774 <+90>:    sub    %eax,%edx
0x0000555555554776 <+92>:    mov    %edx,%eax
0x0000555555554778 <+94>:    mov    %eax,%edx
0x000055555555477a <+96>:    add    %edx,%edx
0x000055555555477c <+98>:    add    %eax,%edx
0x000055555555477e <+100>:   mov    %ecx,%eax
0x0000555555554780 <+102>:   sub    %edx,%eax
0x0000555555554782 <+104>:   test   %eax,%eax
0x0000555555554784 <+106>:   je     0x55555555479e <main+132>
```

test 와 cmp 의 차이

continue구문을 살펴보며 test명령어가 나온것을 확인 할 수 있었다.
test명령어도 cmp처럼 비교구문에 사용되는것 처럼보이는데 차이는 다음과 같다.

CMP : 두개의 operand의 차이를 이용해 flag비트를 설정하여 비교

TEST : 두개의 operand를 서로 AND 연산해 flag비트를 설정하여 비교

배열

지금까지 변수는 특정한 데이터 타입을 저장하는 메모리공간이였다. 그렇다면 변수가 여러개인 배열은 어떻게 표현될까?

```c
#include <stdio.h>

int main(void)
{
    int array[10] = {0};
    int i;

    for(i = 0; i < 10; i++)
    {
        array[i] = i;
    }

    return 0;
}
```

크기가 10인 int형 배열 array를 선언하여 어셈블리를 분석해보자.

배열

이전 슬라이드 c코드를 보면 배열 선언후 내부 값을 0으로 초기화해주는 부분을 확인 할 수 있다. 이와 관련해서 어셈블리어로 보면 변수들의 주소가 연속적으로 할당된것을 볼수있다.

아래 코드를 보면 %rbp-0x8 ~ %rbp-0x30까지 40바이트가 0으로 초기화되었다.
마지막의 movl $0x0,-0x34(%rbp)는 변수 i의 메모리주소이다.

```
0x000055555554681 <+23>:    movq   $0x0,-0x30(%rbp)
0x000055555554689 <+31>:    movq   $0x0,-0x28(%rbp)
0x000055555554691 <+39>:    movq   $0x0,-0x20(%rbp)
0x000055555554699 <+47>:    movq   $0x0,-0x18(%rbp)
0x0000555555546a1 <+55>:    movq   $0x0,-0x10(%rbp)
0x0000555555546a9 <+63>:    movl   $0x0,-0x34(%rbp)
```

배열

이전 슬라이드 c코드를 보면 배열 선언후 내부 값을 0으로 초기화해주는 부분을 확인 할 수 있다. 이와 관련해서 어셈블리어로 보면 변수들의 주소가 연속적으로 할당된것을 볼수있다.

아래 코드를 보면 %rbp-0x8 ~ %rbp-0x30까지 40바이트가 0으로 초기화되었다.
마지막의 movl $0x0,-0x34(%rbp)는 변수 i의 메모리주소이다.

```
0x000055555554681 <+23>:    movq   $0x0,-0x30(%rbp)
0x000055555554689 <+31>:    movq   $0x0,-0x28(%rbp)
0x000055555554691 <+39>:    movq   $0x0,-0x20(%rbp)
0x000055555554699 <+47>:    movq   $0x0,-0x18(%rbp)
0x0000555555546a1 <+55>:    movq   $0x0,-0x10(%rbp)
0x0000555555546a9 <+63>:    movl   $0x0,-0x34(%rbp)
```

배열

```
(gdb) x $rbp
0x7fffffffdf20: 0x555546f0
(gdb) x $rbp-0x30
0x7fffffffdef0: 0x00000000
(gdb) x $rbp-0x2C
0x7fffffffdef4: 0x00000001
(gdb) x $rbp-0x28
0x7fffffffdef8: 0x00000002
(gdb) x $rbp-0x24
0x7fffffffdefc: 0x00000003
(gdb) x $rbp-0x20
0x7fffffffdf00: 0x00000004
(gdb) x $rbp-0x1C
0x7fffffffdf04: 0x00000005
(gdb) x $rbp-0x18
0x7fffffffdf08: 0x00000006
(gdb) x $rbp-0x14
0x7fffffffdf0c: 0x00000007
(gdb) x $rbp-0x10
0x7fffffffdf10: 0x00000008
(gdb) x $rbp-0x0C
0x7fffffffdf14: 0x00000009
```

그러면 전 슬라이드에서 살펴본 주솟값들을 쭉 적어보자

$rbp = 0x7fffffffdf20
$rbp – 0x30(array[0]) = 0x7fffffffdef0
$rbp – 0x2C(array[1]) = 0x7fffffffdef4
$rbp – 0x28(array[2]) = 0x7fffffffdef8
$rbp – 0x24(array[3]) = 0x7fffffffdefc
$rbp – 0x20(array[4]) = 0x7fffffffdf00
$rbp – 0x1C(array[5]) = 0x7fffffffdf04
$rbp – 0x18(array[6]) = 0x7fffffffdf08
$rbp – 0x14(array[7]) = 0x7fffffffdf0c
$rbp – 0x10(array[8]) = 0x7fffffffdf10
$rbp – 0x0C(array[9]) = 0x7fffffffdf14

배열

내용을 정리하자면 배열의 주소값은 연속적으로 할당 되어있는것을 확인 할 수 있다.

지금까지는 array[index]형식으로 사용한 것을 확인 할 수 있는데 만약 그냥 array는 어떻게 될까?

아래의 그림을 보면 array 값과 array[0]의 주소값이 같은것을 확인할수 있다.
이로보아 array는 배열의 시작주소라는것을 알 수 있다.

```
array[0]의 주소 : 0xb33b7db0
array[0]의 값 : 0
array : 0xb33b7db0
```

이중배열

아래의 c코드처럼 이중배열을 선언할 수 있다. 이제 이중 배열의 주소 배치를 알아보자.

```c
#include <stdio.h>

int main(void)
{
    int array[2][2] = {0};
    int i,j;
    int num = 1;

    for(i = 0; i < 2; i++)
    {
        for(j = 0; j < 2; j++)
        {
            array[i][j] = num;
            num++;
        }
    }

    return 0;
}
```

이중배열

아래 어셈블리 코드를 보면 <+23>라인과 <+31> 라인에서 배열을 0으로 초기화하는 것을 확인할 수 있다.

여기서 주소를 살펴보면 %rbp-0x10 ~ %rbp-0x20 16바이트는 앞에서 선언한 배열의 주소이고

%rbp-0x28와 %rbp-0x2C은 변수i,j가 저장되어있는 주소이다.
그렇다면 실제 어느 주소에 어떻게 저장되어있는지 살펴보자.

```
0x0000555555554681 <+23>:    movq   $0x0,-0x20(%rbp)
0x0000555555554689 <+31>:    movq   $0x0,-0x18(%rbp)
0x0000555555554691 <+39>:    movl   $0x1,-0x24(%rbp)
0x0000555555554698 <+46>:    movl   $0x0,-0x2c(%rbp)
0x000055555555469f <+53>:    jmp    0x5555555546d4 <
0x00005555555546a1 <+55>:    movl   $0x0,-0x28(%rbp)
```

이중배열

```
(gdb) x $rbp
0x7fffffffdf30: 0x55554700
(gdb) x $rbp-0x20
0x7fffffffdf10: 0x00000001
(gdb) x $rbp-0x1C
0x7fffffffdf14: 0x00000002
(gdb) x $rbp-0x18
0x7fffffffdf18: 0x00000003
(gdb) x $rbp-0x14
0x7fffffffdf1c: 0x00000004
```

그러면 전 슬라이드에서 살펴본 주솟값들을 쭉 적어보자

$rbp = 0x7fffffffdf30
$rbp – 0x20(array[0][0]) = 0x7fffffffdf10
$rbp – 0x1C(array[0][1]) = 0x7fffffffdf14
$rbp – 0x18(array[1][0]) = 0x7fffffffdf18
$rbp – 0x14(array[1][1]) = 0x7fffffffdf1C

배열정리

0	1	2	3	4	5	6	7	8	9

위의 배열은 배열을 살펴봤을때의 배열이다.

	0	1
0	1	2
1	3	4

위의 배열은 이중배열을 살펴봤을때의 배열이다.

지금까지 분석한것을 정리하자면 배열은 일정한 데이터타입의 데이터가 연속적으로 메모리에 저장되어있는 것을 알수 있었으며, 이중배열 역시 배열과 똑같았다.

Goto문 분석

프로그램 실행중에 goto를 만나면 지정된 레이블로 무조건 분기한다.

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(void)
{
    int i, j, k, random;

    printf("딥러닝 연산중임\n");
    printf("Tensor 기반 연산이다보니 시간이 for 루프가 많음\n");
    printf("하드웨어 장치에서 데이터를 가져와서 처리하고 있음\n");
    printf("그런데 데이터가 갑자기 누락되어서 연산 자체를 폐기해야 하는 상황임\n");
    printf("우리가 실제 에러를 만들 순 없으니 특정한 순간을 가정하고 진행함\n");

    // 난수 생성을 위한 시드값 초기화
    srand(time(NULL));
    random = rand() % 5;

    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 3; j++)
        {
            for (k = 0; k < 3; k++)
            {
                if (j == random)
                {
                    printf("data: %3d, Error!\n", random);
                    goto err_handler;
                }

                printf("i = %3d, j = %3d, k = %d\n", i, j, k);
            }
        }
    }

    printf("정상 종료되었습니다.\n");

    return 0;

err_handler:
    printf("에러가 발생했습니다.\n");
    // -2는 데이터 수집을 못했다는 에러 코드라 가정
    return -2;
}
```

C코드를 보면 삼중 for문이 구현되어있는것을 볼 수 있는데 삼중 for문은 시간복잡도가 O(n^3)이 걸리는 연산이 매우많은 로직이다.

이러한 삼중for문의 동작중에 에러가 발생했을때 goto 문을 이용하여 빠져나오는 코드이다.

Goto문 분석

goto문을 어셈블리로 분석해보면 goto문을 만나면 에러 print를 출력하고 바로 main문의 return 으로 빠지는것을 볼 수 있다.

```
0x0000555555554812 <+184>:   callq  0x555555554600 <printf@plt>
0x0000555555554817 <+189>:   nop
0x0000555555554818 <+190>:   lea    0x28f(%rip),%rdi        # 0x
0x000055555555481f <+197>:   callq  0x5555555545f0 <puts@plt>
0x0000555555554824 <+202>:   mov    $0xfffffffe,%eax
0x0000555555554829 <+207>:   jmp    0x555555554876 <main+284>
```

```
0x0000555555554876 <+284>:   leaveq
0x0000555555554877 <+285>:   retq
```

Goto성능 비교

goto문을 사용했을 때와 사용하지 않았을 때의 성능차이를 비교해보자.

이전슬라이드의 코드와 거의 유사하며 차이점은 for문의 루프 횟수를 100x100x100 = 1000000번한다는 것과 에러발생시 goto를 사용 유무에 차이를 뒀다.

이에따른 실행시간을 비교해 보자.

```
for (i = 0; i < 100; i++)
{
    for (j = 0; j < 100; j++)
    {
        for (k = 0; k < 100; k++)
        {
            if (j == random)
            {
                printf("data: %3d, Error!\n", random);
                goto err_handler;
            }

            printf("i = %3d, j = %3d, k = %d\n", i, j, k);
        }
    }
}
```

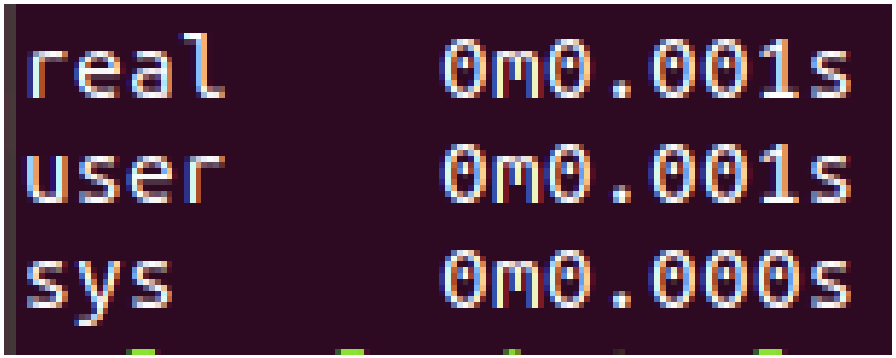
```
for (i = 0; i < 100; i++)
{
    for (j = 0; j < 100; j++)
    {
        for (k = 0; k < 100; k++)
        {
            if (j == random)
            {
                printf("data: %3d, Error!\n", random);
                //goto err_handler;
            }

            printf("i = %3d, j = %3d, k = %d\n", i, j, k);
        }
    }
}
```

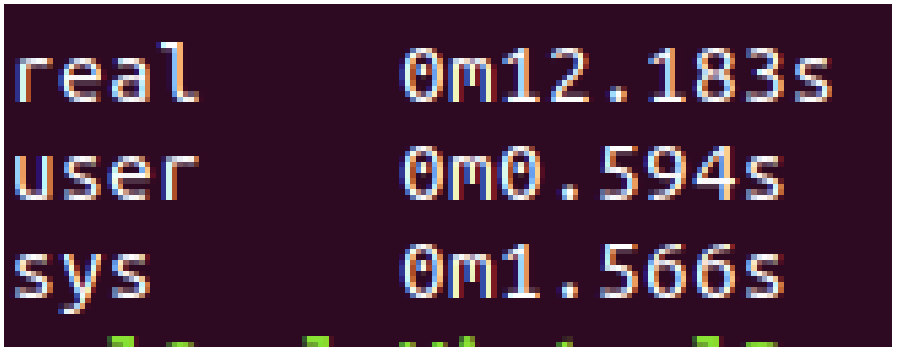
Goto성능 비교

왼쪽의 시간은 goto를 사용한 프로그램의 실행시간이고 오른쪽은 에러가발생해도 그대로 실행두게 놔두 프로그램의 실행시간이다.

Real 부분의 시간을 비교해보면 10000배이상의 실행시간 차이가 나는것을 알수있다.

이로보아 요즘에 핫한 머신러닝의 텐서연산시 goto를 활용하면 실행시간을 단축시킬 수 있다는 것을 알수 있다.





문제

1. 3명의 사원이 있고 각각의 사원에게 2400 ~ 3600 사이의 랜덤한 급여를 지급합니다.
1년마다 연봉 인상률을 1 ~ 10% 적용하여 10년후 각 사원들의 급여를 출력해보도록 합니다.

```
******** 1년차 월급 ********
사원 1 :  2490
사원 2 :  2887
사원 3 :  3261

********* 연봉 인상률 *********
사원 1 : 1%
사원 2 : 10%
사원 3 : 7%
********  2년차 월급 ********
사원 1 :  2514
사원 2 :  3175
사원 3 :  3489

********* 연봉 인상률 *********
사원 1 : 6%
사원 2 : 5%
사원 3 : 8%
********  3년차 월급 ********
사원 1 :  2664
사원 2 :  3333
사원 3 :  3768

********* 연봉 인상률 *********
사원 1 : 2%
사원 2 : 4%
사원 3 : 8%
********  4년차 월급 ********
사원 1 :  2717
사원 2 :  3466
사원 3 :  4069

********* 연봉 인상률 *********
사원 1 : 1%
사원 2 : 3%
사원 3 : 3%
********  5년차 월급 ********
사원 1 :  2744
사원 2 :  3569
사원 3 :  4191
```

```
********* 연봉 인상률 *********
사원 1 : 6%
사원 2 : 2%
사원 3 : 6%
********  6년차 월급 ********
사원 1 :  2908
사원 2 :  3640
사원 3 :  4442

********* 연봉 인상률 *********
사원 1 : 1%
사원 2 : 1%
사원 3 : 4%
********  7년차 월급 ********
사원 1 :  2937
사원 2 :  3676
사원 3 :  4619

********* 연봉 인상률 *********
사원 1 : 5%
사원 2 : 8%
사원 3 : 3%
********  8년차 월급 ********
사원 1 :  3083
사원 2 :  3970
사원 3 :  4757

********* 연봉 인상률 *********
사원 1 : 7%
사원 2 : 9%
사원 3 : 1%
********  9년차 월급 ********
사원 1 :  3298
사원 2 :  4327
사원 3 :  4804
```

```
********* 연봉 인상률 *********
사원 1 : 1%
사원 2 : 9%
사원 3 : 8%
******** 10년차 월급 ********
사원 1 :  3330
사원 2 :  4716
사원 3 :  5188

********* 연봉 인상률 *********
사원 1 : 4%
사원 2 : 2%
사원 3 : 6%
******** 11년차 월급 ********
사원 1 :  3463
사원 2 :  4810
사원 3 :  5499
```

문제

2. 컴퓨터와 주사위를 굴려서 숫자가 높은 사람이 이기도록 프로그래밍 해봅시다.

```
컴퓨터 주사위 숫자   : 1
플레이어 주사위 숫자 : 3

플레이어 승리!!
eyl@eyl-VirtualBox:~/proj/Lv01-02/HyunhoCha/homework/c/05$ ./homework5
컴퓨터 주사위 숫자   : 5
플레이어 주사위 숫자 : 6

플레이어 승리!!
eyl@eyl-VirtualBox:~/proj/Lv01-02/HyunhoCha/homework/c/05$ ./homework5
컴퓨터 주사위 숫자   : 3
플레이어 주사위 숫자 : 1
```

문제

3. 주사위를 3개 굴려서 숫자가 높은 사람이 이기도록 만들어봅시다.
   (주사위 눈금을 모두 더한 결과를 비교한다)

```
컴퓨터 주사위 숫자   : 5, 6, 6
플레이어 주사위 숫자 : 3, 4, 1

컴퓨터 승리!!
eyl@eyl-VirtualBox:~/proj/Lv01-02/HyunhoCha/homework/c/05$ ./homework5
컴퓨터 주사위 숫자   : 1, 4, 6
플레이어 주사위 숫자 : 2, 3, 4

컴퓨터 승리!!
eyl@eyl-VirtualBox:~/proj/Lv01-02/HyunhoCha/homework/c/05$ ./homework5
컴퓨터 주사위 숫자   : 4, 6, 2
플레이어 주사위 숫자 : 2, 6, 1

컴퓨터 승리!!
```

문제

4. 조금 더 확장해서 3명의 플레이어가 주사위를 가지고 게임을 한다 가정합니다.
   주사위를 2개 굴려서 숫자가 높은 사람이 이기게 해봅시다.
   (마찬가지로 주사위 눈금을 모두 더해 비교하도록 한다)

```
플레이어1 주사위 숫자 : 5, 5
플레이어2 주사위 숫자 : 1, 5
플레이어3 주사위 숫자 : 2, 3

플레이어1승리!!
eyl@eyl-VirtualBox:~/proj/Lv01-02/HyunhoCha/homework/c/05$ ./homework5
플레이어1 주사위 숫자 : 3, 3
플레이어2 주사위 숫자 : 6, 2
플레이어3 주사위 숫자 : 3, 5

플레이어2, 플레이어3 승리!!
eyl@eyl-VirtualBox:~/proj/Lv01-02/HyunhoCha/homework/c/05$ ./homework5
플레이어1 주사위 숫자 : 3, 4
플레이어2 주사위 숫자 : 5, 3
플레이어3 주사위 숫자 : 3, 5

플레이어2, 플레이어3 승리!!
```

문제

5. 주사위 게임에 추가적인 옵션 기능을 구현합니다.
1번째 주사위가 홀수면 주사위를 1개만 굴릴 수 있습니다.
-만약 짝수가 나오면 2번째 주사위를 굴릴 수 있습니다.

2번째 주사위는 옵션이 있는데 숫자 4가 나오면 0점 처리됩니다.
-숫자 3이 나온 경우엔 1번째 굴린 주사위를 무조건 2배수 처리합니다.
-숫자 6이 나온 경우엔 자신을 제외한 모든 플레이어들의 주사위 숫자를 -4 처리합니다.
-숫자 1이 나오면 지정한 사람의 주사위를 +3 할 수 있고 추가로 또 한 사람을 지명하여 -2 할 수 있습니다.

문제

```
--------- 1번째 주사위 ---------
플레이어1 주사위 숫자 : 5
플레이어2 주사위 숫자 : 3
플레이어3 주사위 숫자 : 2

플레이어1 점수 : 5
플레이어2 점수 : 3
플레이어3 점수 : 2

--------- 2번째 주사위 숫자 ---------
플레이어1 주사위 숫자 : 0
플레이어2 주사위 숫자 : 0
플레이어3 주사위 숫자 : 4

플레이어1 점수 : 5
플레이어2 점수 : 3
플레이어3 점수 : 0

플레이어1 승리!!
```

```
--------- 1번째 주사위 ---------
플레이어1 주사위 숫자 : 3
플레이어2 주사위 숫자 : 5
플레이어3 주사위 숫자 : 5

플레이어1 점수 : 3
플레이어2 점수 : 5
플레이어3 점수 : 5

--------- 2번째 주사위 숫자 ---------
플레이어1 주사위 숫자 : 0
플레이어2 주사위 숫자 : 0
플레이어3 주사위 숫자 : 0

플레이어1 점수 : 3
플레이어2 점수 : 5
플레이어3 점수 : 5

플레이어2, 플레이어3 승리!!
```

```
--------- 1번째 주사위 ---------
플레이어1 주사위 숫자 : 5
플레이어2 주사위 숫자 : 4
플레이어3 주사위 숫자 : 5

플레이어1 점수 : 5
플레이어2 점수 : 4
플레이어3 점수 : 5

--------- 2번째 주사위 숫자 ---------
플레이어1 주사위 숫자 : 0
플레이어2 주사위 숫자 : 5
플레이어3 주사위 숫자 : 0

플레이어1 점수 : 5
플레이어2 점수 : 9
플레이어3 점수 : 5

플레이어2 승리!!
```

질문