C Programming - 5

임베디드스쿨2기

Lv2과정

2020. 04. 23

박태인

1. 배열 (1)

1) 배열을 사용하는 이유

- int 형 변수가 1000개 필요하면 ???
  - ㄴ 일일히 int a, b, c, d, e, f, g, h, ... zzz 까지 적기도 힘들다.
  - ㄴ 여러개의 변수를 한번에 선언한다.

- 배열 선언 방법
  1. 먼저 다발로 **활용할 데이터 타입**을 적는다.
  2. 변수명이 있듯이 **배열의 이름**을 적는다.
  3. **얼마만큼의 공간**을 활용할지 **숫자를 대괄호 내부에** 적는다.
     (여기서 대괄호를 비워두면 입력되는 요소에 따라 자동으로 개수가 정해진다)
     입력이 없을 경우엔 문제가 될 수 있으니
     필요한 개수를 설정해놓는 것을 권장한다.
     혹은 입력할 데이터를 미리 설정해놓는것도 좋다.

- 선언 예시

```
// arr
// [0]   [1]   [2]
//  1     2     3
// 배열의 시작 인덱스는 0부터 시작하므로 주의해야 한다.
// 선언할 때는 사용할 개수를 적지만
// 활용할 때는 적은 개수 - 1까지 활용이 가능하다는 것을 주의하라!
int arr[] = { 2, 4, 7 };
```

1. 배열 (2)

```c
int arr[] = { 2, 4, 7 };
int len = sizeof(arr) / sizeof(int);

printf("arr len = %d\n", len);

printf("arr:\n");

for (i = 0; i < len; i++)
{
        printf("%2d", arr[i]);
}

printf("\n");

return 0;
```

2, 4, 7 값을 가지는 int 형 배열을 선언한다.
배열의 길이는 배열 크기 / 배열 형태로 구한다.

배열의 길이를 프린트하고,

For 문을 통해 배열 길이 만큼 반복문을 실행해서

배열의 값을 프린트 한다.

```
arr len = 3
arr:
 2 4 7
```

1. 배열 (3)

자, 이번에는 앞서 했던 배열을 어셈블리로 분석해 보자.

```
   0x00005555555551a9 <+0>:     endbr64
   0x00005555555551ad <+4>:     push   %rbp
=> 0x00005555555551ae <+5>:     mov    %rsp,%rbp
   0x00005555555551b1 <+8>:     sub    $0x20,%rsp
   0x00005555555551b5 <+12>:    mov    %fs:0x28,%rax
```

자.. 이제 si를 실행해서 push %rbp명령어를 실행 했습니다.
Push 명령어는 **현재 스택의 최상위 메모리(rsp)**에 값을 저장하는 명령어 입니다.
즉, **현재 스택의 최상위 메모리(rsp)**에 **rbp값을 저장하라**는 의미 겠죠.
그런데, 초기 rbp의 값은 0x0 이었으므로 아래와 같이 구성 되겠습니다.

0x0 (rbp)	0x0x7fffffffdfa0 (rsp)

```
(gdb) x $rbp
0x0:    Cannot
(gdb) x $rsp
0x7fffffffdfa0:
```

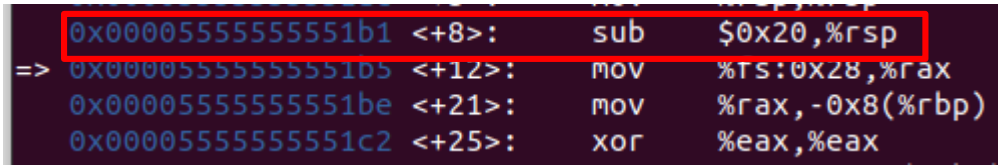
```
   0x00005555555551a9 <+0>:     endbr64
   0x00005555555551ad <+4>:     push   %rbp
=> 0x00005555555551ae <+5>:     mov    %rsp,%rbp
   0x00005555555551b1 <+8>:     sub    $0x20,%rsp
   0x00005555555551b5 <+12>:    mov    %fs:0x28,%rax
```

```
(gdb) x $rbp
0x7fffffffdfa0:
(gdb) x $rsp
0x7fffffffdfa0:
```

자 이제 한번 더 si를 실행해서 mov %rsp, %rbp 를 시행 해봅시다.
Mov 명령어는 내용을 복사하는 것 입니다.
즉, mov rsp rbp는 rbp에 rsp 값을 복사합니다. (rsp 값 → rbp 값)
ㄴ일반적인 A = B 꼴에서 B 값이 A로 들어가는 것이 아닌 반대 방향으로 생각해야 하는 것에 주의 하자!

그러면 어떻게 되겠나요?
ㄴ 결국, rbp에 rsp 값을 넣어 버림 으로써 **rsp, rbp 주소 값이 서로 같아지면서 스택의 경계선이 사라집니다!**
→ 이것은 **새로운 스택을 생성 할 준비를 하는 과정** 입니다.

1. 배열 (4)

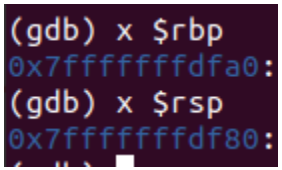


자 이번에는 sub 명령어를 실행 했습니다.
Sub 명령어는 뺄셈 명령 입니다.
Sub 0x20, rsp는 현재 rsp 에서 32**바이트를 빼겠다는 의미** 입니다. (0x20 → 0010 0000 : 2^5 = 32 **바이트,**
가상 주소 공간에서는 바이트 단위로 움직입니다!)

그림으로 나타내어 보면 아래와 같습니다.

0x0(rbp)	0x0x7fffffffdfa0 ← rbp
	<<— sub로 만든 공간이 지역 변수 공간으로 쓰는 메모리 공간
	0x0x7fffffffdf80 ← rsp

ㄴ 이런 구조 때문에 스택은 아래로 자란다고 한 것이다!!



: 현재 스택의 최상위 rsp 80
현재 스택의 기준점 rbp a0

1. 배열 (5)

```
   0x00005555555551b5 <+12>:    mov    %fs:0x28,%rax
   0x00005555555551be <+21>:    mov    %rax,-0x8(%rbp)
   0x00005555555551c2 <+25>:    xor    %eax,%eax
   0x00005555555551c4 <+27>:    movl   $0x2,-0x14(%rbp)
   0x00005555555551cb <+34>:    movl   $0x4,-0x10(%rbp)
   0x00005555555551d2 <+41>:    movl   $0x7,-0xc(%rbp)
   0x00005555555551d9 <+48>:    movl   $0x3,-0x18(%rbp)
   0x00005555555551e0 <+55>:    mov    -0x18(%rbp),%eax
   0x00005555555551e3 <+58>:    mov    %eax,%esi
   0x00005555555551e5 <+60>:    lea    0xe18(%rip),%rdi        # 0x555555556004
   0x00005555555551ec <+67>:    mov    $0x0,%eax
```

```
int arr[] = { 2, 4, 7 };
```

ㄴ C 원문의 배열 선언

- Mov %fs:0x28, %rax
  ㄴ 보안상의 코드를 rax에 배치
- mov %rax, -0x8(%rbp)
  ㄴ rax 값을 rbp-8 에 배치
- xor %eax, %eax
  ㄴ xor의 특징은 값이 같으면 0이 나오죠?
  이런 특징을 이용해서 eax를 0으로
  초기화 했다는 걸 알 수 있습니다.
- movl $0x2, -0x14(%rbp)
  ㄴ 2의 값을 rbp-20에 배치
- movl $0x4, -0x10(%rbp)
  ㄴ 4의 값은 rbp-16에 배치
- movl $0x7, -0xc (%rbp)
  ㄴ 7의 값을 rbp-12에 배치
- movl $0x3, -0x18 (%rbp)
  ㄴ 3의 값을 rbp-24에 배치
-mov -0x18(%rbp), %eax
  ㄴ rbp-24의 값인 3을 eax에 복사
- mov %eax, %esi
  ㄴ eax값을 esi에 복사
- lea 0xe18(%rip), %rdi
  ㄴ 배열의 값을 rdi에 배치
- mov 0x0, %eax
  ㄴ eax를 0으로 초기화.

```
(gdb) x $rbp-8
0x7fffffffdf98: 0x693dc100
(gdb) x $rax
0x9415030e693dc100:     Can
(gdb) x $rbp-4
0x7fffffffdf9c: 0x9415030e
```

```
(gdb) p/x $eax
$2 = 0x0
```

```
(gdb) x $rbp-20
0x7fffffffdf8c: 0x00000002
(gdb) x $rbp-16
0x7fffffffdf90: 0x00000004
(gdb) x $rbp-12
0x7fffffffdf94: 0x00000007
(gdb) x $rbp-24
0x7fffffffdf88: 0x00000003
```

```
(gdb) x $eax
0x3:    Canno
(gdb) x $esi
0x3:    Canno
(gdb) si
0x00005555555
(gdb) x $eax
0x0:    Canno
```

0x0(rbp)	0x7fffffffdfa0 ← rbp
%rax	0x7fffffffdf98 ← rbp-8
7	0x7fffffffdf94 ← rbp-12
4	0x7fffffffdf90 ← rbp-16
2	0x7fffffffdf8c ← rbp-20
3	0x7fffffffdf88 ← rbp-24
	0x7fffffffdf80 ← rsp

1. 배열 (6)

```
=> 0x00005555555551f1 <+72>:    callq  0x5555555550b0 <printf@plt>
   0x00005555555551f6 <+77>:    lea    0xe15(%rip),%rdi        # 0x555555556012
   0x00005555555551fd <+84>:    callq  0x555555555090 <puts@plt>
   0x0000555555555202 <+89>:    movl   $0x0,-0x1c(%rbp)
   0x0000555555555209 <+96>:    jmp    0x55555555522b <main+130>
   0x000055555555520b <+98>:    mov    -0x1c(%rbp),%eax
```

ㄴ callq는 중요한 문구 이므로, 다시한번 되새겨 보면
이 명령어는 기본적으로 push + jmp 인데,
복귀주소를 저장하고, jump를 한다.
Priintf 다음 함수에서는 다음 명령어인 lea의 주소값인 1f6 값을
rsp-8 에 저장하게 된다.

```
(gdb) x $rsp-8
0x7fffffffdf78: 0x555551f6
```

- lea 0xe15(%rip), %rdi
  ㄴ 배열 값을 rdi에 배치

```
(gdb) x $rdi
0x555555556012: 0x3a727261
```

- 다음 callq 의 puts 명령어 또한 다음 복귀 주소를 rsp-8에 배치 후 jmp

```
(gdb) x $rsp-8
0x7fffffffdf78: 0x55555202
```

- movl $0x0, -0x1c(%rbp)
  ㄴ 0 값을 rbp-28에 배치.

- jmp 0x55...22b , 22b 위치로 점프

```
(gdb) si
0x000055555555522b      32              for (i = 0; i < len; i++)
```

0x0(rbp)	0x7fffffffdfa0 ← rbp
%rax	0x7fffffffdf98 ← rbp-8
7	0x7fffffffdf94 ← rbp-12
4	0x7fffffffdf90 ← rbp-16
2	0x7fffffffdf8c ← rbp-20
3	0x7fffffffdf88 ← rbp-24
0	0x7fffffffdf84 ← rbp-28
	0x7fffffffdf80 ← rsp
0x55555202(복귀)	0x7fffffffdf78 ← rsp-8

1. 배열 (7)

```
=> 0x000055555555522b <+130>:    mov    -0x1c(%rbp),%eax
   0x000055555555522e <+133>:    cmp    -0x18(%rbp),%eax
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000555555555231 <+136>:    jl     0x55555555520b <main+98>
   0x0000555555555233 <+138>:    mov    $0xa,%edi
   0x0000555555555238 <+143>:    callq  0x555555555080 <putchar@plt>
   0x000055555555523d <+148>:    mov    $0x0,%eax
   0x0000555555555242 <+153>:    mov    -0x8(%rbp),%rdx
   0x0000555555555246 <+157>:    xor    %fs:0x28,%rdx
```

- mov -0x1c(%rbp), %eax
  - ㄴ rbp-28 값인 0 을 eax에 복사
- cmp -0x18(%rbp), %eax
  - ㄴ rbp-24 값 (3) 과 eax 값 (0) 을 비교
- jl 0x555...20b
  - ㄴ 비교대상보다 비교값이 더 작으므로 jmp

```
(gdb) x $eax
0x0:    Cannot access memo
(gdb) x $rbp-24
0x7fffffffdf88: 0x00000003
```

```
=> 0x000055555555520b <+98>:    mov    -0x1c(%rbp),%eax
   0x000055555555520e <+101>:   cltq
   0x0000555555555210 <+103>:   mov    -0x14(%rbp,%rax,4),%eax
```

0x0(rbp)	0x7fffffffdfa0 ← rbp
%rax	0x7fffffffdf98 ← rbp-8
7	0x7fffffffdf94 ← rbp-12
4	0x7fffffffdf90 ← rbp-16
2	0x7fffffffdf8c ← rbp-20
3	0x7fffffffdf88 ← rbp-24
0	0x7fffffffdf84 ← rbp-28
	0x7fffffffdf80 ← rsp
0x55555202(복귀)	0x7fffffffdf78 ← rsp-8

1. 배열 (8)

```
=> 0x000055555555520b <+98>:    mov    -0x1c(%rbp),%eax
   0x000055555555520e <+101>:   cltq
   0x0000555555555210 <+103>:   mov    -0x14(%rbp,%rax,4),%eax
   0x0000555555555214 <+107>:   mov    %eax,%esi
   0x0000555555555216 <+109>:   lea    0xdfa(%rip),%rdi        # 0x555555556017
   0x000055555555521d <+116>:   mov    $0x0,%eax
   0x0000555555555222 <+121>:   callq  0x5555555550b0 <printf@plt>
```

- mov -0x1c(%rbp), %eax
  ㄴrbp-28 값인 0 을 eax에 복사
- mov -0x14(%rbp, %rax, 4), %eax
  ㄴ rbp-20 값인 2를 eax에 복사

```
(gdb) x $eax
0x2:    Cann
```

- mov %eax, %esi
  ㄴ eax 값인 2를 esi에 복사
- lea 0xdfa(%rip), %rdi
  ㄴ 배열 주소에 값을 rdi 에 배치
- mov 0x0, %eax
  ㄴ eax를 다시 0 값으로 초기화

```
(gdb) x $eax
0x2:    Cannot access memor
(gdb) x $esi
0x2:    Cannot access memor
(gdb) x $rdi
0x555555556017: 0x00643225
```

- callq 0x555… pritf
  ㄴ rsp-8에 복귀주소를 저장 한 뒤 jmp

```
=> 0x0000555555555227 <+126>:   addl   $0x1,-0x1c(%rbp)
   0x000055555555522b <+130>:   mov    -0x1c(%rbp),%eax
   0x000055555555522e <+133>:   cmp    -0x18(%rbp),%eax
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000555555555231 <+136>:   jl     0x55555555520b <main+98>
   0x0000555555555233 <+138>:   mov    $0xa,%edi
   0x0000555555555238 <+143>:   callq  0x555555555080 <putchar@plt>
```

- addl $0x1, -0x1c(%rbp)
  ㄴ rbp-28 값에 1 을 더한다.

0x0(rbp)	0x7fffffffdfa0 ← rbp
%rax	0x7fffffffdf98 ← rbp-8
7	0x7fffffffdf94 ← rbp-12
4	0x7fffffffdf90 ← rbp-16
2	0x7fffffffdf8c ← rbp-20
3	0x7fffffffdf88 ← rbp-24
0 +1	0x7fffffffdf84 ← rbp-28
	0x7fffffffdf80 ← rsp
0x55555227(복귀)	0x7fffffffdf78 ← rsp-8

1. 배열 (9)

```
=> 0x0000555555555227 <+126>:   addl   $0x1,-0x1c(%rbp)
   0x000055555555522b <+130>:   mov    -0x1c(%rbp),%eax
   0x000055555555522e <+133>:   cmp    -0x18(%rbp),%eax
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000555555555231 <+136>:   jl     0x55555555520b <main+98>
   0x0000555555555233 <+138>:   mov    $0xa,%edi
   0x0000555555555238 <+143>:   callq  0x555555555080 <putchar@plt>
```

- mov -0x1c(%rbp), %eax
  ㄴ rbp – 28 값 (1) 을 eax 에 복사
- cmp -0x18(%rbp), %eax
  ㄴ rbp – 24 값 (3) 을 eax (1) 와 비교
- jl
  ㄴ 비교대상보다 비교 값이 작으면 jmp (20b)

```
(gdb) x $eax
0x1:    Canno
```

```
int arr[] = { 2, 4, 7 };
int len = sizeof(arr) / sizeof(int);

printf("arr len = %d\n", len);

printf("arr:\n");

for (i = 0; i < len; i++)
{
        printf("%2d", arr[i]);
}

printf("\n");

return 0;
```

- 이런 식으로 원문의 for 문을 조건을 비교하고 조건이 일치하면 For 문을 계속해서 동작하게 된다.

0x0(rbp)	0x7fffffffdfa0 ← rbp
%rax	0x7fffffffdf98 ← rbp-8
7	0x7fffffffdf94 ← rbp-12
4	0x7fffffffdf90 ← rbp-16
2	0x7fffffffdf8c ← rbp-20
3	0x7fffffffdf88 ← rbp-24
0 +1	0x7fffffffdf84 ← rbp-28
	0x7fffffffdf80 ← rsp
0x55555227(복귀)	0x7fffffffdf78 ← rsp-8

1. 배열 (10)

```
   0x0000555555555231 <+136>:   jl     0x55555555520b <main+98>
=> 0x0000555555555233 <+138>:   mov    $0xa,%edi
   0x0000555555555238 <+143>:   callq  0x555555555080 <putchar@plt>
   0x000055555555523d <+148>:   mov    $0x0,%eax
   0x0000555555555242 <+153>:   mov    -0x8(%rbp),%rdx
   0x0000555555555246 <+157>:   xor    %fs:0x28,%rdx
   0x000055555555524f <+166>:   je     0x555555555256 <main+173>
   0x0000555555555251 <+168>:   callq  0x5555555550a0 <__stack_chk_fail@plt>
   0x0000555555555256 <+173>:   leaveq
   0x0000555555555257 <+174>:   retq
```

ㄴ for 문의 비교문이 끝나고 나면 위와 같이 다음으로 넘어간다.

- mov $0xa, %edi
  ㄴ 0xa의 값을 edi 에 배치

```
(gdb) x $edi
0xa:    Cann
```

- callq 명령어
  ㄴ 복귀 주소를 rsp-8에 push 후 점프

- mov $0x0, $eax
  ㄴ eax 값을 0으로 초기화
- mov -0x8(rbp), %rdx
  ㄴ rbp-8 값을 rdx에 배치
- xor %fs.. : 보안상의 명령어, xor이므로 비교 값이 다르면 1 아니면 0
- je : equal jump (xor 해서 0 값 만들어서 jmp)
- leaveq : 스택해제명령
- retq : 돌아갈 함수가 있었다면 리턴

```
(gdb) x $rdx
0x0:    Canno
(gdb) x $eax
0x0:    Canno
```

```
   0x000055555555524f <+166>:   je     0x555555555256 <main+173>
   0x0000555555555251 <+168>:   callq  0x5555555550a0 <__stack_chk_fail@plt>
=> 0x0000555555555256 <+173>:   leaveq
   0x0000555555555257 <+174>:   retq
```

0x0(rbp)	0x7fffffffdfa0 ← rbp
%rax	0x7fffffffdf98 ← rbp-8
7	0x7fffffffdf94 ← rbp-12
4	0x7fffffffdf90 ← rbp-16
2	0x7fffffffdf8c ← rbp-20
3	0x7fffffffdf88 ← rbp-24
0 +1+1+1	0x7fffffffdf84 ← rbp-28
	0x7fffffffdf80 ← rsp
0x5555523d(복귀)	0x7fffffffdf78 ← rsp-8

2. Continue

```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;
}
```

```
1 ~ n까지 출력합니다. (n을 선택하세요): 9
i =   1
i =   2
i =   4
i =   5
i =   7
i =   8
```

Continue 명령어는 반복문을
실행하지 않고 다시 조건으로 돌아가는
명령어 이다.

왼쪽의 코딩을 보면
n항의 숫자로 입력 받고,

그 항 만큼 for 문을 통해 반복을 하게 된다.

그 와중에 if문은 3의 배수가 나오면
참 값이 되어 들어가게 되고

continue를 만나 아래의
printf는 실행하지 않게 됩니다.

2. Continue(2)

자, 이번에는 앞서 했던 배열을 어셈블리로 분석해 보자.

```
=> 0x0000555555555189 <+0>:     endbr64
   0x000055555555518d <+4>:     push   %rbp
   0x000055555555518e <+5>:     mov    %rsp,%rbp
   0x0000555555555191 <+8>:     sub    $0x10,%rsp
   0x0000555555555195 <+12>:    mov    %fs:0x28,%rax
```

자.. 이제 si를 실행해서 push %rbp명령어를 실행 했습니다.
Push 명령어는 **현재 스택의 최상위 메모리(rsp)**에 값을 저장하는 명령어 입니다.
즉, **현재 스택의 최상위 메모리(rsp)**에 **rbp값을 저장하라**는 의미 겠죠.
그런데, 초기 rbp의 값은 0x0 이었으므로 아래와 같이 구성 되겠습니다.

0x0 (rbp)	0x0x7fffffffdf90 (rsp)

```
(gdb) x $rbp
0x0:    Cannot access memory
(gdb) x $rsp
0x7fffffffdf90: 0x00000000
```

```
   0x00005555555551a9 <+0>:     endbr64
   0x00005555555551ad <+4>:     push   %rbp
=> 0x00005555555551ae <+5>:     mov    %rsp,%rbp
   0x00005555555551b1 <+8>:     sub    $0x20,%rsp
   0x00005555555551b5 <+12>:    mov    %fs:0x28,%rax
```

```
(gdb) x $rbp
0x7fffffffdf90:
(gdb) x $rsp
0x7fffffffdf90:
```

자 이제 한번 더 si를 실행해서 mov %rsp, %rbp 를 시행 해봅시다.
Mov 명령어는 내용을 복사하는 것 입니다.
즉, mov rsp rbp는 rbp에 rsp 값을 복사합니다. (rsp 값 → rbp 값)
ㄴ일반적인 A = B 꼴에서 B 값이 A로 들어가는 것이 아닌 반대 방향으로 생각해야 하는 것에 주의 하자!

그러면 어떻게 되겠나요?
ㄴ 결국, rbp에 rsp 값을 넣어 버림 으로써 **rsp, rbp 주소 값이 서로 같아지면서 스택의 경계선이 사라집니다!**
→ 이것은 **새로운 스택을 생성 할 준비를 하는 과정** 입니다.

2. Continue(3)

```
=> 0x0000555555555191 <+8>:     sub    $0x10,%rsp
   0x0000555555555195 <+12>:    mov    %fs:0x28,%rax
   0x000055555555519e <+21>:    mov    %rax,-0x8(%rbp)
   0x00005555555551a2 <+25>:    xor    %eax,%eax
```

자 이번에는 sub 명령어를 실행 했습니다.
Sub 명령어는 뺄셈 명령 입니다.
Sub 0x10, rsp는 현재 rsp 에서 16**바이트를 빼겠다는 의미** 입니다. (0x20 → 0010 0000 : 2^4 = 16 **바이트,**
가상 주소 공간에서는 바이트 단위로 움직입니다!)

그림으로 나타내어 보면 아래와 같습니다.

0x0(rbp)	0x0x7fffffffdf90 ← rbp
	<<— sub로 만든 공간이 지역 변수 공간으로 쓰는 메모리 공간
	0x0x7fffffffdf80 ← rsp

ㄴ 이런 구조 때문에 스택은 아래로 자란다고 한 것이다!!

```
(gdb) x $rbp
0x7fffffffdf90:
(gdb) x $rsp
0x7fffffffdf80:
```

: 현재 스택의 최상위 rsp 80
현재 스택의 기준점 rbp 90

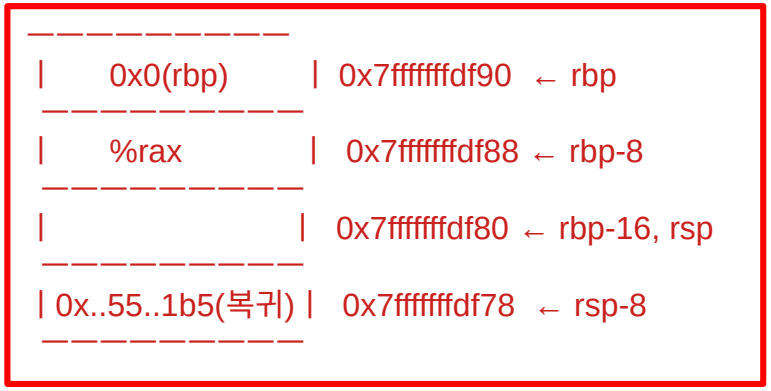
2. Continue(4)

```
=> 0x0000555555555195 <+12>:    mov    %fs:0x28,%rax
   0x000055555555519e <+21>:    mov    %rax,-0x8(%rbp)
   0x00005555555551a2 <+25>:    xor    %eax,%eax
   0x00005555555551a4 <+27>:    lea    0xe5d(%rip),%rdi        # 0x555555556008
   0x00005555555551ab <+34>:    mov    $0x0,%eax
   0x00005555555551b0 <+39>:    callq  0x555555555080 <printf@plt>
```

- Mov %fs:0x28, %rax
  - ㄴ 보안상의 코드를 rax에 배치
- mov %rax, -0x8(%rbp)
  - ㄴ rax 값을 rbp-8 에 배치
- xor %eax, %eax
  - ㄴ xor의 특징은 값이 같으면 0이 나오죠?
    이런 특징을 이용해서 eax를 0으로
    초기화 했다는 걸 알 수 있습니다.

- lea 0xe5d(%rip), %rdi
  - ㄴ 배열의 값을 rdi에 배치
- mov 0x0, %eax
  - ㄴ eax를 0으로 초기화.

- callq 복귀 주소를 rsp-8 에 저장 후 printf 함수로 진입.




```
(gdb) x $rsp-8
0x7fffffffdf78: 0x555551b5
```

```
   0x00005555555551b0 <+39>:    callq  0x555555555080 <printf@plt>
=> 0x00005555555551b5 <+44>:    lea    -0x10(%rbp),%rax
```

2. Continue(5)

```
=> 0x00005555555551b5 <+44>:    lea    -0x10(%rbp),%rax
   0x00005555555551b9 <+48>:    mov    %rax,%rsi
   0x00005555555551bc <+51>:    lea    0xe7b(%rip),%rdi        # 0x55555555603e
   0x00005555555551c3 <+58>:    mov    $0x0,%eax
   0x00005555555551c8 <+63>:    callq  0x555555555090 <__isoc99_scanf@plt>
   0x00005555555551cd <+68>:    movl   $0x1,-0xc(%rbp)
```

- lea -0x10(%rbp), %rax
  ㄴ rbp-16 위치의 값을 rax에 배치한다.

```
(gdb) x $rbp-16
0x7fffffffdf80: 0xffffe080
(gdb) x $rax
0x7fffffffdf80: 0xffffe080
```

- mov %rax, %rsi
  ㄴ rax 값을 %rsi에 복사한다.

```
(gdb) x $rax
0x7fffffffdf80: 0xffffe080
(gdb) x $rsi
0x7fffffffdf80: 0xffffe080
```

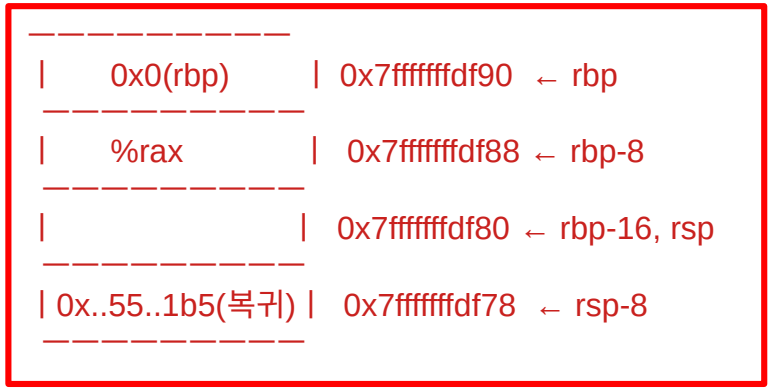
- lea 0xe7b(%rip), %rdi
  ㄴ 배열의 값을 rdi에 배치

- mov 0x0, %eax
  ㄴ eax를 0으로 초기화.

- callq 복귀 주소를 rsp-8 에 저장 후 scanf 함수로 진입.

```
(gdb) ni
1 ~ n까지 출력합니다. (n을 선택하세요): 3
```

ㄴ n 에 3을 입력.



2. Continue(6)

```
=> 0x00005555555551cd <+68>:    movl   $0x1,-0xc(%rbp)
   0x00005555555551d4 <+75>:    jmp    0x555555555220 <main+151>
```

- movl $0x1, -0xc(rbp)
  ㄴ 0x1의 값을 rbp-12에 복사.

```
(gdb) x $rbp-12
0x7fffffffdf84: 0x00000001
```

- jmp 0x55...220 으로 점프

```
=> 0x0000555555555220 <+151>:   mov    -0x10(%rbp),%eax
   0x0000555555555223 <+154>:   cmp    %eax,-0xc(%rbp)
   0x0000555555555226 <+157>:   jle    0x5555555551d6 <main+77>
   0x0000555555555228 <+159>:   mov    $0x0,%eax
   0x000055555555522d <+164>:   mov    -0x8(%rbp),%rdi
   0x0000555555555231 <+168>:   xor    %fs:0x28,%rdi
   0x000055555555523a <+177>:   je     0x555555555241 <main+184>
   0x000055555555523c <+179>:   callq  0x555555555070 <__stack_chk_fail@plt>
   0x0000555555555241 <+184>:   leaveq
   0x0000555555555242 <+185>:   retq
```

- Mov -0x10(%rbp), %eax
  ㄴ rbp-16의 값을 eax에 복사.
- %eax, -0xc(%rbp)
  ㄴeax 값(3)과 rbp-12 값(1) 을 비교.
- jle
  ㄴ3보다 작거나 같으면 점프 이므로 1d6로 점프

0x0(rbp)	0x7fffffffdf90 ← rbp
%rax	0x7fffffffdf88 ← rbp-8
1	0x7fffffffdf84 ← rbp-12
3	0x7fffffffdf80 ← rbp-16, rsp
0x..55..1b5(복귀)	0x7fffffffdf78 ← rsp-8

2. Continue(7)

```
=> 0x00005555555551d6 <+77>:    mov    -0xc(%rbp),%ecx
   0x00005555555551d9 <+80>:    movslq %ecx,%rax
   0x00005555555551dc <+83>:    imul   $0x55555556,%rax,%rax
   0x00005555555551e3 <+90>:    shr    $0x20,%rax
   0x00005555555551e7 <+94>:    mov    %rax,%rdx
   0x00005555555551ea <+97>:    mov    %ecx,%eax
   0x00005555555551ec <+99>:    sar    $0x1f,%eax
   0x00005555555551ef <+102>:   mov    %edx,%esi
   0x00005555555551f1 <+104>:   sub    %eax,%esi
   0x00005555555551f3 <+106>:   mov    %esi,%eax
   0x00005555555551f5 <+108>:   mov    %eax,%edx
   0x00005555555551f7 <+110>:   add    %edx,%edx
   0x00005555555551f9 <+112>:   add    %eax,%edx
   0x00005555555551fb <+114>:   mov    %ecx,%eax
   0x00005555555551fd <+116>:   sub    %edx,%eax
   0x00005555555551ff <+118>:   test   %eax,%eax
```

0x0(rbp)	0x7fffffffdf90 ← rbp
%rax	0x7fffffffdf88 ← rbp-8
1	0x7fffffffdf84 ← rbp-12
3	0x7fffffffdf80 ← rbp-16, rsp
0x..55..1b5(복귀)	0x7fffffffdf78 ← rsp-8

- Mov -0xc(%rbp), %ecx
  ㄴ rbp-12 값(1)을 ecx에 복사.
- movslq $ecx, %rax
  ㄴ ecx 값(1)을 rax 에 복사.(1)
- **imul → 부호가 있는 곱셈**
  ㄴ rax(1) 곱하기 rax(1)
- **shr → shift right (부호가 없는 연산) : 2배로 나누기**
  ㄴ 1만큼 오른쪽 쉬프트 하고 rax에 저장
  → 얼만큼 쉬**프트 시킬지 숫자가 정해지지 않으면 1만큼 이동이다.**
- mov %rax, %rdx
  ㄴ rax값을 rdx에 복사
- mov $ecx, %eax
  ㄴ ecx값을 eax에 복사.
- **sar $0x1f, %eax**, sar → **shift rihgt(부호가 있는 연산)**
  ㄴ 1만큼 오른쪽 쉬프트 하고 eax에 저장
- mov %edx, %esi
  ㄴ edx 값을 esi에 복사.

2. Continue(8)

```
=> 0x00005555555551d6 <+77>:    mov    -0xc(%rbp),%ecx
   0x00005555555551d9 <+80>:    movslq %ecx,%rax
   0x00005555555551dc <+83>:    imul   $0x55555556,%rax,%rax
   0x00005555555551e3 <+90>:    shr    $0x20,%rax
   0x00005555555551e7 <+94>:    mov    %rax,%rdx
   0x00005555555551ea <+97>:    mov    %ecx,%eax
   0x00005555555551ec <+99>:    sar    $0x1f,%eax
   0x00005555555551ef <+102>:   mov    %edx,%esi
   0x00005555555551f1 <+104>:   sub    %eax,%esi
   0x00005555555551f3 <+106>:   mov    %esi,%eax
   0x00005555555551f5 <+108>:   mov    %eax,%edx
   0x00005555555551f7 <+110>:   add    %edx,%edx
   0x00005555555551f9 <+112>:   add    %eax,%edx
   0x00005555555551fb <+114>:   mov    %ecx,%eax
   0x00005555555551fd <+116>:   sub    %edx,%eax
   0x00005555555551ff <+118>:   test   %eax,%eax
```

0x0(rbp)	0x7fffffffdf90 ← rbp
%rax	0x7fffffffdf88 ← rbp-8
1	0x7fffffffdf84 ← rbp-12
3	0x7fffffffdf80 ← rbp-16, rsp
0x..55..1b5(복귀)	0x7fffffffdf78 ← rsp-8

- sub %eax, %esi
  ㄴ eax = eax – esi
- mov %esi, %eax
  ㄴ esi 값 eax에 복사.
- mov %eax, %edx
  ㄴ eax값 edx에 복사.
- add %edx, %edx
  ㄴ edx = edx + edx
- add %eax, %edx
  ㄴ eax = eax + edx
- mov %ecx, %eax
  ㄴ ecx 값을 eax 에 복사
- sub %edx, %eax
  ㄴ edx = edx-eax
- test %eax, %eax
  ㄴ **test는 두 값을 AND 시켜서 두 값이 모두 0인지를 확인한다.**
  ㄴ **if 문 조건 성립 여부 확인**

```
if (!(i % 3))
```

2. Continue(9)

```
   0x0000555555555201 <+120>:   je     0x55555555521b <main+146>
=> 0x0000555555555203 <+122>:   mov    -0xc(%rbp),%eax
   0x0000555555555206 <+125>:   mov    %eax,%esi
   0x0000555555555208 <+127>:   lea    0xe32(%rip),%rdi        # 0x555555556041
   0x000055555555520f <+134>:   mov    $0x0,%eax
   0x0000555555555214 <+139>:   callq  0x555555555080 <printf@plt>
   0x0000555555555219 <+144>:   jmp    0x55555555521c <main+147>
```

- 앞선 ppt에서 if 조건문이 거짓 jump 되지 않고 위와 같이 아래로 진행
- mov -0xc(%rbp), %eax
  ㄴ rbp-12 값을 eax에 복사.
- mov %eax, %esi
  ㄴ eax 값을 esi에 복사
- lea 0xe32($rip), %rdi
  ㄴ 배열 값을 rdi에 배치
- mov $0x0, %eax
  ㄴ eax 값을 0 으로 초기화
- callq 복귀 주소를 rsp-8 에 push 하고 printf 점프.

```
(gdb) x $rsp-8
0x7fffffffdf78: 0x55555219
```

0x0(rbp)	0x7fffffffdf90 ← rbp
%rax	0x7fffffffdf88 ← rbp-8
1	0x7fffffffdf84 ← rbp-12
3	0x7fffffffdf80 ← rbp-16, rsp
0x..55..219(복귀)	0x7fffffffdf78 ← rsp-8

=> **<u>if 문의 continue가 실행되지 않고, for 문의 printf를 실행하게 되는 과정이다.</u>**

- jmp 0x55...21c 로 점프 → for 문을 다시 돌게됨.

```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;
}
```

2. Continue(10)

```
=> 0x000055555555521c <+147>:   addl   $0x1,-0xc(%rbp)
   0x0000555555555220 <+151>:   mov    -0x10(%rbp),%eax
   0x0000555555555223 <+154>:   cmp    %eax,-0xc(%rbp)
   0x0000555555555226 <+157>:   jle    0x5555555551d6 <main+77>
   0x0000555555555228 <+159>:   mov    $0x0,%eax
   0x000055555555522d <+164>:   mov    -0x8(%rbp),%rdi
   0x0000555555555231 <+168>:   xor    %fs:0x28,%rdi
   0x000055555555523a <+177>:   je     0x555555555241 <main+184>
   0x000055555555523c <+179>:   callq  0x555555555070 <__stack_chk_fail@plt>
```

- addl $0x1, -0xc(%rbp)
  ㄴ rbp-12 값에 1을 더한다.
- mov -0x10(%rbp), %eax
  ㄴ rbp-16 값을 eax에 배치
- cmp %eax, -0xc(%rbp)
  ㄴ eax값과 rbp-12 값을 비교
- jle 1d6
  ㄴ 비교대상보다 비교값이 작거나 같으면 1d6으로 jmp
  → if 조건문으로 다시 가게 되고, if 조건문을 다시 한번 실행하게 된다.
  → rbp-12 값이 아직 3이 아니므로 if 문으로 들어가지 않고 위의 과정을 다시 반복하게 된다.

0x0(rbp)	0x7fffffffdf90	← rbp
%rax	0x7fffffffdf88	← rbp-8
1 +1	0x7fffffffdf84	← rbp-12
3	0x7fffffffdf80	← rbp-16, rsp
0x..55..219(복귀)	0x7fffffffdf78	← rsp-8

```
   0x0000555555555201 <+120>:   je     0x55555555521b <main+146>
=> 0x0000555555555203 <+122>:   mov    -0xc(%rbp),%eax
   0x0000555555555206 <+125>:   mov    %eax,%esi
   0x0000555555555208 <+127>:   lea    0xe32(%rip),%rdi        # 0x55
   0x000055555555520f <+134>:   mov    $0x0,%eax
   0x0000555555555214 <+139>:   callq  0x555555555080 <printf@plt>
   0x0000555555555219 <+144>:   jmp    0x55555555521c <main+147>
```

- 이 부분에서 다시 한번 je 조건문에서 점프 하지 못하고 아래 명령문으로 가서 Printf 함수를 동작하게 된다.

```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;
}
```

2. Continue(11)

```
=> 0x000055555555521c <+147>:   addl   $0x1,-0xc(%rbp)
   0x0000555555555220 <+151>:   mov    -0x10(%rbp),%eax
   0x0000555555555223 <+154>:   cmp    %eax,-0xc(%rbp)
   0x0000555555555226 <+157>:   jle    0x5555555551d6 <main+77>
   0x0000555555555228 <+159>:   mov    $0x0,%eax
   0x000055555555522d <+164>:   mov    -0x8(%rbp),%rdi
   0x0000555555555231 <+168>:   xor    %fs:0x28,%rdi
   0x000055555555523a <+177>:   je     0x555555555241 <main+184>
   0x000055555555523c <+179>:   callq  0x555555555070 <__stack_chk_fail@plt>
```

- 위와 같이 다시 for문을 돌고 if문의 조건을 확인한다.
- 그리고 드디어 if문에 입성하게 된다.

```
(gdb) si
16                              continue;
```

- if 문의 cotinue로 가게 된다.

```
=> 0x000055555555521b <+146>:   nop
   0x000055555555521c <+147>:   addl   $0x1,-0xc(%rbp)
   0x0000555555555220 <+151>:   mov    -0x10(%rbp),%eax
   0x0000555555555223 <+154>:   cmp    %eax,-0xc(%rbp)
   0x0000555555555226 <+157>:   jle    0x5555555551d6 <main+77>
   0x0000555555555228 <+159>:   mov    $0x0,%eax
   0x000055555555522d <+164>:   mov    -0x8(%rbp),%rdi
   0x0000555555555231 <+168>:   xor    %fs:0x28,%rdi
   0x000055555555523a <+177>:   je     0x555555555241 <main+184>
   0x000055555555523c <+179>:   callq  0x555555555070 <__stack_chk_fail@plt>
   0x0000555555555241 <+184>:   leaveq
   0x0000555555555242 <+185>:   retq
```

- nop은 아무명령을 하지 않고 쉬어 가는 것이다.
- addl $0x1, -0xc(%rbp)
  ㄴ rpb-12 의 값에 1을 더한다.
- cmp %eax, -0xc(%rbp)
  ㄴ eax와 rbp-12 값을 비교하고
- jle
  ㄴ 비교 대상보다 작거나 같으면 점프(작거나 같지 않으므로 점프하지 않는다
  : for문 종료)

=> **continue를 실행 했더니**
아래 printf문을 실행하지 않고
For 조건문으로 다시 돌아 간 것이다!

0x0(rbp)	0x7fffffffdf90 ← rbp
%rax	0x7fffffffdf88 ← rbp-8
1 +1 +1	0x7fffffffdf84 ← rbp-12
3	0x7fffffffdf80 ← rbp-16, rsp
0x..55..219(복귀)	0x7fffffffdf78 ← rsp-8

```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;
}
```

2. Continue(12)

```
=> 0x0000555555555228 <+159>:   mov    $0x0,%eax
   0x000055555555522d <+164>:   mov    -0x8(%rbp),%rdi
   0x0000555555555231 <+168>:   xor    %fs:0x28,%rdi
   0x000055555555523a <+177>:   je     0x555555555241 <main+184>
   0x000055555555523c <+179>:   callq  0x555555555070 <__stack_chk_fail@plt>
   0x0000555555555241 <+184>:   leaveq
   0x0000555555555242 <+185>:   retq
```

- mov $0x0, %eax
  ㄴ 0 값을 eax에 배치
- mov -0x8(%rbp), %rdi
  ㄴ rbp-8 값을 rdi에 배치
- xor %fs..
  ㄴ 보안상의 코드
- je 0x55..241
  ㄴ 241로 점프
- leaveq
  ㄴ 스택해제 명령.

0x0(rbp)	0x7fffffffdf90 ← rbp
%rax	0x7fffffffdf88 ← rbp-8
1 +1 +1	0x7fffffffdf84 ← rbp-12
3	0x7fffffffdf80 ← rbp-16, rsp
0x..55..219(복귀)	0x7fffffffdf78 ← rsp-8

```
   0x0000555555555228 <+159>:   mov    $0x0,%eax
   0x000055555555522d <+164>:   mov    -0x8(%rbp),%rdi
   0x0000555555555231 <+168>:   xor    %fs:0x28,%rdi
   0x000055555555523a <+177>:   je     0x555555555241 <main+184>
   0x000055555555523c <+179>:   callq  0x555555555070 <__stack_chk_fail@plt>
=> 0x0000555555555241 <+184>:   leaveq
   0x0000555555555242 <+185>:   retq
```

For 문이 종료 되어

Return 0; 실행.

```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;
}
```

3. goto (1)

```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() % 3;

        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);
                                }

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

        return 0;
}
```

```
딥러닝 연산중임
Tensor 기반 연산이다보니 시간이 for 루프가 많음
하드웨어 장치에서 데이터를 가져와서 처리하고 있음
그런데 데이터가 갑자기 누락되어서 연산 자체를 폐기해야 하는 상황임
우리가 실제 에러를 만들 순 없으니 특정한 순간을 가정하고 진행함
i =   0, j =   0, k = 0
i =   0, j =   0, k = 1
i =   0, j =   0, k = 2
i =   0, j =   1, k = 0
i =   0, j =   1, k = 1
i =   0, j =   1, k = 2
data:   2, Error!
i =   0, j =   2, k = 0
data:   2, Error!
i =   0, j =   2, k = 1
```

Goto 문을 사용해 보기 전에
왼쪽과 같이 조건을

난수 함수 srand(time(NULL));
Rand(); 를 사용하여
조건을 만들고

for문속에 if문을 조건을
임의의 Error 조건으로 완성하여
Error 발생이 랜덤적으로 생기게
해 본다.

여기서

난수 생성을 위한 함수
srand와 rand에 대해서 좀 더 조사해
보자(다음 페이지)

3. 이중 배열 (1)

이중배열에서 주의 할 점은
이중 배열은 실제 **이들은 차원을 가지고 있지 않다는 점**이다.
이중 배열은 순차적으로 배치 되어 있게 된다.

예를 들어 **int arr[2][2]** 라는 배열이 있다면

[0]	[1]
[0][0] [0][1]	[1][0] [1][1]

또 다른 예로 **int arr[3][3]**

[0]	[1]	[2]
[0][0] [0][1] [0][2]	[1][0] [1][1] [1][2]	[2][0] [2][1] [2][2]

하나더 하면 int **arr[2][4]**

[0]	[1]
[0][0] [0][1] [0][2] [0][3]	[1][0] [1][1] [1][2] [1][3]

3. 이중 배열 (2)

```
int i, j;
// 이중 배열
// 실제 이들은 차원을 가지고 있지 않으며 순차적으로 배치되어 있다.

//      [0]          [1]
// [0][0] [0][1] [1][0] [1][1]
//   0      20     10     30
int arr[2][2];

//          [0]                 [1]                 [2]
// [0][0] [0][1] [0][2] [1][0] [1][1] [1][2] [2][0] [2][1] [2][2]
int arr2[3][3];

//             [0]                         [1]
// [0][0] [0][1] [0][2] [0][3] [1][0] [1][1] [1][2] [1][3]
int arr3[2][4];
```

```
for (i = 0; i < 2; i++)
{
        for (j = 0; j < 2; j++)
        {
                arr[i][j] = i * 10 + j * 20;
                printf("arr[%d][%d] = %d\n", i, j, arr[i][j]);
        }
}

return 0;
```

arr[i][j]
의 배열 구조를 for문 두개를 만들어서
출력하는 구조 이다.

위의 int arr[2][2]의 구조에
각각의 값이 0 20 10 30 이 들어 가게 된다.

```
arr[0][0] = 0
arr[0][1] = 20
arr[1][0] = 10
arr[1][1] = 30
```

4. 더블 포인터

```c
#include <stdio.h>

int main(void)
{
        int num = 3;
        int *p_num = &num;
        int **pp_num = &p_num;

        // p7_num;

        printf("num = %d\n", num);
        printf("*p_num = %d\n", *p_num);
        printf("**pp_num = %d\n", **pp_num);

        printf("&num = 0x%x\n", &num);
        printf("p_num = 0x%x\n", p_num);

        printf("&p_num = 0x%x\n", &p_num);
        printf("pp_num = 0x%x\n", pp_num);

        printf("&pp_num = 0x%x\n", &pp_num);

        return 0;
}
```

```
num = 3
*p_num = 3
**pp_num = 3
&num = 0xb835694
p_num = 0xb835694
&p_num = 0xb835698
pp_num = 0xb835698
&pp_num = 0xb8356a0
```

Num = 3 초기화
***p_num은 num 변수의 주소 값**
****pp_num은 p_num 포인터 변수의 주소값**이다.

printf의 *p_num은 p_num 포인터 변수가 가르키는 주소의 값을 출력한다.

마찬가지로 **p_num은 p_num의 포인터 변수가 가르키는 주소의 값 = num의 주소 값 이므로 다시 한번 들어가면 num이 가르키는 값을 출력하게 된다.