EDDI
Electronic Design
Development Institute

에디로봇아카데미

임베디드 마스터 Lv1 과정

제 4기

2022. 10. 27

진동민

학습목표 & 5회차 날짜

EDDI
Electronic Design
Development Institute

학습목표

- 배열을 어셈블리어로 분석하기
- 포인터 변수와 변수의 차이 이해하기
- 포인터 변수 참조 방법 이해하기
- AND NOT 기법 이해하기
- 2차원 배열을 반환하는 함수 프로토타입 이해하기

수업 날짜

2022-09-24 (토) 오후 6시~9시

목차

배열 선언의 어셈블리어 분석

EDDI
Electronic Design Development Institute

1) array.c

```c
#include <stdio.h>

int main(void)
{
    // 배열의 선언 방법
    // 1. 데이터 타입 작성 (int, float, double 등등)
    // 2. 변수 선언하듯이 이름 작성
    // 3. 대괄호 열고 몇 개를 만들지 숫자를 작성
    // 4. 필요하다면 { } 중괄호 내부에 배치할 값들을 작성

    int array[5] = { 1, 2, 3, 4 };
    int i;

    for (i = 0; i < 5; i++)
    {
        printf("array[%d] = %d\n", i, array[i]);
    }

    return 0;
}
```

carbon
carbon.now.sh

배열의 목적

일괄적인 관리를 위해

배열 선언의 어셈블리어 분석

EDDI
Electronic Design Development Institute

2) array.c의 어셈블리어

```
(gdb) disas
Dump of assembler code for function main:
=> 0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   %rbp
   0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x30,%rsp
   0x0000555555555175 <+12>:    mov    %fs:0x28,%rax
   0x000055555555517e <+21>:    mov    %rax,-0x8(%rbp)
   0x0000555555555182 <+25>:    xor    %eax,%eax
   0x0000555555555184 <+27>:    movq   $0x0,-0x20(%rbp)
   0x000055555555518c <+35>:    movq   $0x0,-0x18(%rbp)
   0x0000555555555194 <+43>:    movl   $0x0,-0x10(%rbp)
   0x000055555555519b <+50>:    movl   $0x1,-0x20(%rbp)
   0x00005555555551a2 <+57>:    movl   $0x2,-0x1c(%rbp)
   0x00005555555551a9 <+64>:    movl   $0x3,-0x18(%rbp)
   0x00005555555551b0 <+71>:    movl   $0x4,-0x14(%rbp)
   0x00005555555551b7 <+78>:    movl   $0x0,-0x24(%rbp)
   0x00005555555551be <+85>:    jmp    0x5555555551e3 <main+122>
   0x00005555555551c0 <+87>:    mov    -0x24(%rbp),%eax
   0x00005555555551c3 <+90>:    cltq
   0x00005555555551c5 <+92>:    mov    -0x20(%rbp,%rax,4),%edx
   0x00005555555551c9 <+96>:    mov    -0x24(%rbp),%eax
   0x00005555555551cc <+99>:    mov    %eax,%esi
   0x00005555555551ce <+101>:   lea    0xe2f(%rip),%rdi        # 0x555555556004
   0x00005555555551d5 <+108>:   mov    $0x0,%eax
   0x00005555555551da <+113>:   callq  0x555555555070 <printf@plt>
   0x00005555555551df <+118>:   addl   $0x1,-0x24(%rbp)
   0x00005555555551e3 <+122>:   cmpl   $0x4,-0x24(%rbp)
   0x00005555555551e7 <+126>:   jle    0x5555555551c0 <main+87>
   0x00005555555551e9 <+128>:   mov    $0x0,%eax
   0x00005555555551ee <+133>:   mov    -0x8(%rbp),%rcx
   0x00005555555551f2 <+137>:   xor    %fs:0x28,%rcx
   0x00005555555551fb <+146>:   je     0x5555555555202 <main+153>
   0x00005555555551fd <+148>:   callq  0x555555555060 <__stack_chk_fail@plt>
   0x0000555555555202 <+153>:   leaveq
   0x0000555555555203 <+154>:   retq
End of assembler dump.
```

배열 선언의 어셈블리어 분석

3) array.c의 어셈블리어 중요 포인트 분석

```
(gdb) disas
Dump of assembler code for function main:
=> 0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   %rbp
   0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x30,%rsp
   0x0000555555555175 <+12>:    mov    %fs:0x28,%rax
   0x000055555555517e <+21>:    mov    %rax,-0x8(%rbp)
   0x0000555555555182 <+25>:    xor    %eax,%eax
   0x0000555555555184 <+27>:    movq   $0x0,-0x20(%rbp)
   0x000055555555518c <+35>:    movq   $0x0,-0x18(%rbp)
   0x0000555555555194 <+43>:    movl   $0x0,-0x10(%rbp)
   0x000055555555519b <+50>:    movl   $0x1,-0x20(%rbp)
   0x00005555555551a2 <+57>:    movl   $0x2,-0x1c(%rbp)
   0x00005555555551a9 <+64>:    movl   $0x3,-0x18(%rbp)
   0x00005555555551b0 <+71>:    movl   $0x4,-0x14(%rbp)
   0x00005555555551b7 <+78>:    movl   $0x0,-0x24(%rbp)
   0x00005555555551be <+85>:    jmp    0x5555555551e3 <main+122>
   0x00005555555551c0 <+87>:    mov    -0x24(%rbp),%eax
   0x00005555555551c3 <+90>:    cltq
   0x00005555555551c5 <+92>:    mov    -0x20(%rbp,%rax,4),%edx
   0x00005555555551c9 <+96>:    mov    -0x24(%rbp),%eax
   0x00005555555551cc <+99>:    mov    %eax,%esi
   0x00005555555551ce <+101>:   lea    0xe2f(%rip),%rdi        # 0x555555556004
   0x00005555555551d5 <+108>:   mov    $0x0,%eax
   0x00005555555551da <+113>:   callq  0x555555555070 <printf@plt>
   0x00005555555551df <+118>:   addl   $0x1,-0x24(%rbp)
   0x00005555555551e3 <+122>:   cmpl   $0x4,-0x24(%rbp)
   0x00005555555551e7 <+126>:   jle    0x5555555551c0 <main+87>
   0x00005555555551e9 <+128>:   mov    $0x0,%eax
```

← 48바이트 스택 생성

← 배열 할당

← 변수 i 할당

← 배열 요소를 edx에 복사 (자세한 내용은 뒤에서...)

설명

- int array[5] = { 1, 2, 3, 4 };
  - 마지막 요소는 값을 명시해주지 않았으므로 0으로 초기화
- 신기하게도 **배열의 마지막 요소가 먼저 스택에 배치**된 것을 볼 수 있다.

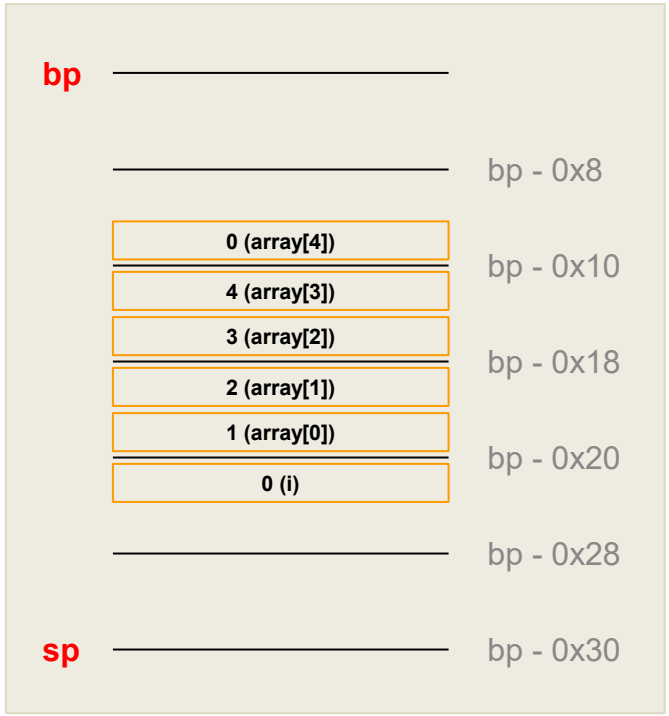
추가 개념

- cltq 명령어는 eax 값을 부호를 포함하여 확장한 값을 rax로 복사

배열 선언의 어셈블리어 분석

4) 앞의 슬라이드에서 '변수 i 할당'까지만 그림으로 그려보자

메모리



설명

- 실제로 스택에 배열의 요소를 배치한 순서는 '0, 1, 2, 3, 4' 순이다.
- 반복문에서 배열의 각 요소에 순차적으로 접근하는 방법
  - 변수 i 값을 eax 레지스터에 저장 후, 배열의 시작 주소에 eax 값과 int형 크기인 4와 곱한 값을 더하여 배열 요소에 접근한다.

```
movl   $0x0,-0x24(%rbp)          ← 변수 i 할당
jmp    0x5555555551e3 <main+122>
mov    -0x24(%rbp),%eax          ← 변수 i를 eax로 복사
cltq
★ mov  -0x20(%rbp,%rax,4),%edx   ← (rbp - 0x20) + rax * 4를 edx로 복사
mov    -0x24(%rbp),%eax
mov    %eax,%esi                 rax는 변수 i 값, 즉 rax 값에 따라 어떤
lea    0xe2f(%rip),%rdi          배열 요소에 접근하는지 달라진다.
mov    $0x0,%eax
callq  0x555555555070 <printf@plt>
addl   $0x1,-0x24(%rbp)          이때 4는 int형의 크기이다.
cmpl   $0x4,-0x24(%rbp)
jle    0x5555555551c0 <main+87>
```

for문

배열 선언의 어셈블리어 분석

EDDI
Electronic Design Development Institute

5) 정리

- 어셈블리어에서 배열의 선언은 변수 여러 개를 선언한 것과 다르지 않은 것을 확인할 수 있었고, 배열 또한 메모리이다.

- 소스코드에서 초기화를 명시해 주지 않은 배열의 요소부터 스택에 먼저 배치한다.
  - 이것이 맞는지 확인하기 위해 배열을 선언할 때의 개수를 6으로 바꾸고 디버깅해보았다.

- for문에서 배열의 각 요소에 순차적으로 접근하는 방법으로 배열의 시작 주소, for문의 i 값 그리고 배열 요소의 자료형 크기와 연산하여 접근하였다.

윈도우에서 나타나는 쓰레기 값은 무엇인가

EDDI
Electronic Design Development Institute

1) 질문: 변수에 쓰레기값이 들어가는데 배열은 아닌가?

이 질문은 수업을 같이 듣는 수강생이 질문했던 것으로 기억한다.

아마 변수를 선언하고 초기화하지 않으면 쓰레기 값이 들어가지만 배열의 초기화의 경우에는 잘 몰랐던 것으로 추측된다.

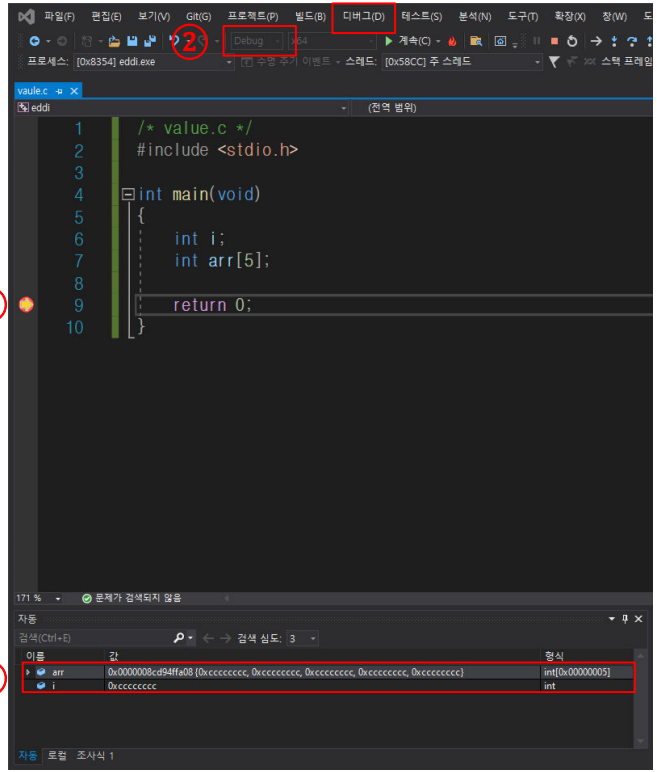
배열을 선언과 동시에 초기화할 경우에 선언한 길이만큼 초기화하지 않으면, 나머지 요소는 0으로 초기화되기 때문이다.

- 예) int arr[5] = { 1, 2, 3, 4 };

그럼 윈도우에서 초기화하지 않은 변수의 쓰레기 값이 무엇인지 알아보자!

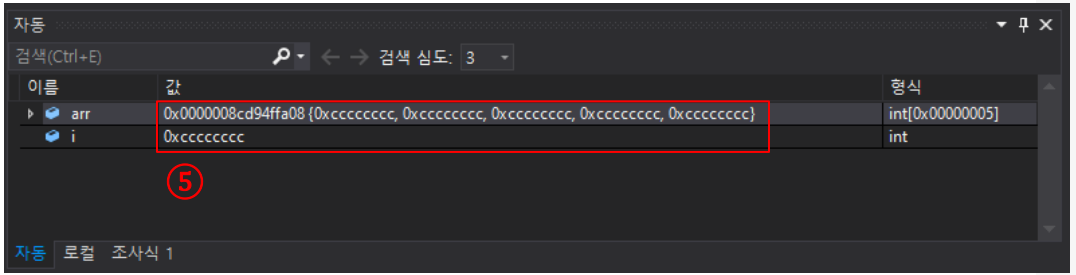
윈도우에서 나타나는 쓰레기 값은 무엇인가

2) 쓰레기 값 확인



방법

1. 코드를 작성하고 return 0;에 중단점을 삽입한다.
1. 컴파일 전 Debug 모드로 설정한다.
1. F5 버튼이나 디버그(D) 메뉴 바에서 디버깅 시작 버튼을 클릭한다.
1. 값을 확인한다.
1. ④에서 값에 마우스 우클릭하여 ‘16진수 표시(H)’를 클릭하면 0xcc 값이 저장된 것을 확인할 수 있다.



윈도우에서 나타나는 쓰레기 값은 무엇인가

3) 그래서 초기화하지 않았을 때의 0xcc 값은 무엇인가

0xcc는 윈도우에서 디버깅을 하는 상황에서 나타나는 값이다.

메모리에 0xcc라는 INT3 인터럽트를 유발하는 것을 채워넣음으로써 해당 영역을 디버깅 하겠다라는 의미이다.

비어있는 변수를 신경쓰지 않고 그냥 싹 다 0xcc로 채워버린 상태라고 생각하면 된다.

출처

- C언어 이슈 사항의 '변수에 나타나는 쓰레기값(0xcccccccc)의 정체는 무엇인가요?' (2022/09/24)

관련 링크

- https://stackoverflow.com/questions/11864817/why-does-the-not-allocated-memory-is-marked-like-0xcc
- https://2ry53.tistory.com/entry/소프트웨어-브레이크-포인터와-하드웨어-브레이크-포인터
- http://daplus.net/debugging-visual-studio-c-에서-메모리-할당-표현은-무엇입니까/

포인터 변수 선언과 참조의 어셈블리어 분석

EDDI
Electronic Design Development Institute

1) pointer.c

```c
#include <stdio.h>

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

    *p = 7;

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

    return 0;
}
```

포인터 변수 선언과 참조의 어셈블리어 분석

EDDI
Electronic Design Development Institute

2) pointer.c의 어셈블리어

```
(gdb) disas
Dump of assembler code for function main:
=> 0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   %rbp
   0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x20,%rsp
   0x0000555555555175 <+12>:    mov    %fs:0x28,%rax
   0x000055555555517e <+21>:    mov    %rax,-0x8(%rbp)
   0x0000555555555182 <+25>:    xor    %eax,%eax
   0x0000555555555184 <+27>:    movl   $0x3,-0x14(%rbp)
   0x000055555555518b <+34>:    lea    -0x14(%rbp),%rax
   0x000055555555518f <+38>:    mov    %rax,-0x10(%rbp)
   0x0000555555555193 <+42>:    mov    -0x10(%rbp),%rax
   0x0000555555555197 <+46>:    movl   $0x7,(%rax)
   0x000055555555519d <+52>:    mov    -0x14(%rbp),%eax
   0x00005555555551a0 <+55>:    mov    %eax,%esi
   0x00005555555551a2 <+57>:    lea    0xe5b(%rip),%rdi        # 0x555555556004
   0x00005555555551a9 <+64>:    mov    $0x0,%eax
   0x00005555555551ae <+69>:    callq  0x555555555070 <printf@plt>
   0x00005555555551b3 <+74>:    mov    -0x10(%rbp),%rax
   0x00005555555551b7 <+78>:    mov    (%rax),%eax
   0x00005555555551b9 <+80>:    mov    %eax,%esi
   0x00005555555551bb <+82>:    lea    0xe4c(%rip),%rdi        # 0x55555555600e
   0x00005555555551c2 <+89>:    mov    $0x0,%eax
   0x00005555555551c7 <+94>:    callq  0x555555555070 <printf@plt>
   0x00005555555551cc <+99>:    mov    $0x0,%eax
   0x00005555555551d1 <+104>:   mov    -0x8(%rbp),%rdx
   0x00005555555551d5 <+108>:   xor    %fs:0x28,%rdx
   0x00005555555551de <+117>:   je     0x55555555551e5 <main+124>
   0x00005555555551e0 <+119>:   callq  0x555555555060 <__stack_chk_fail@plt>
   0x00005555555551e5 <+124>:   leaveq
   0x00005555555551e6 <+125>:   retq
End of assembler dump.
```

포인터 변수 선언과 참조의 어셈블리어 분석

3-1) 포인터 변수를 스택에 어떻게 배치하는지 분석

```
(gdb) disas
Dump of assembler code for function main:
=> 0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   %rbp
   0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x20,%rsp
   0x0000555555555175 <+12>:    mov    %fs:0x28,%rax
   0x000055555555517e <+21>:    mov    %rax,-0x8(%rbp)
   0x0000555555555182 <+25>:    xor    %eax,%eax
   0x0000555555555184 <+27>:    movl   $0x3,-0x14(%rbp)
   0x000055555555518b <+34>:    lea    -0x14(%rbp),%rax
   0x000055555555518f <+38>:    mov    %rax,-0x10(%rbp)
   0x0000555555555193 <+42>:    mov    -0x10(%rbp),%rax
   0x0000555555555197 <+46>:    movl   $0x7,(%rax)
   0x000055555555519d <+52>:    mov    -0x14(%rbp),%eax
   0x00005555555551a0 <+55>:    mov    %eax,%esi
   0x00005555555551a2 <+57>:    lea    0xe5b(%rip),%rdi        # 0x555555556004
   0x00005555555551a9 <+64>:    mov    $0x0,%eax
   0x00005555555551ae <+69>:    callq  0x555555555070 <printf@plt>
```

← 32바이트 스택 생성

← 변수 num 스택에 배치 (int num = 3;)

← 포인터 변수를 스택에 배치 (int* p = #)

포인터 변수 선언과 참조의 어셈블리어 분석

EDDI
Electronic Design Development Institute

3-2) lea 명령어 분석

```
(gdb) disas
Dump of assembler code for function main:
   0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   %rbp
   0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x20,%rsp
   0x0000555555555175 <+12>:    mov    %fs:0x28,%rax
   0x000055555555517e <+21>:    mov    %rax,-0x8(%rbp)
   0x0000555555555182 <+25>:    xor    %eax,%eax
   0x0000555555555184 <+27>:    movl   $0x3,-0x14(%rbp)
   0x000055555555518b <+34>:    lea    -0x14(%rbp),%rax        ← 실행
=> 0x000055555555518f <+38>:    mov    %rax,-0x10(%rbp)
   0x0000555555555193 <+42>:    mov    -0x10(%rbp),%rax
   0x0000555555555197 <+46>:    movl   $0x7,(%rax)
   0x000055555555519d <+52>:    mov    -0x14(%rbp),%eax
   0x00005555555551a0 <+55>:    mov    %eax,%esi
   0x00005555555551a2 <+57>:    lea    0xe5b(%rip),%rdi        # 0x555555556004
   0x00005555555551a9 <+64>:    mov    $0x0,%eax
   0x00005555555551ae <+69>:    callq  0x555555555070 <printf@plt>
```

lea 명령어 실행

(참고)

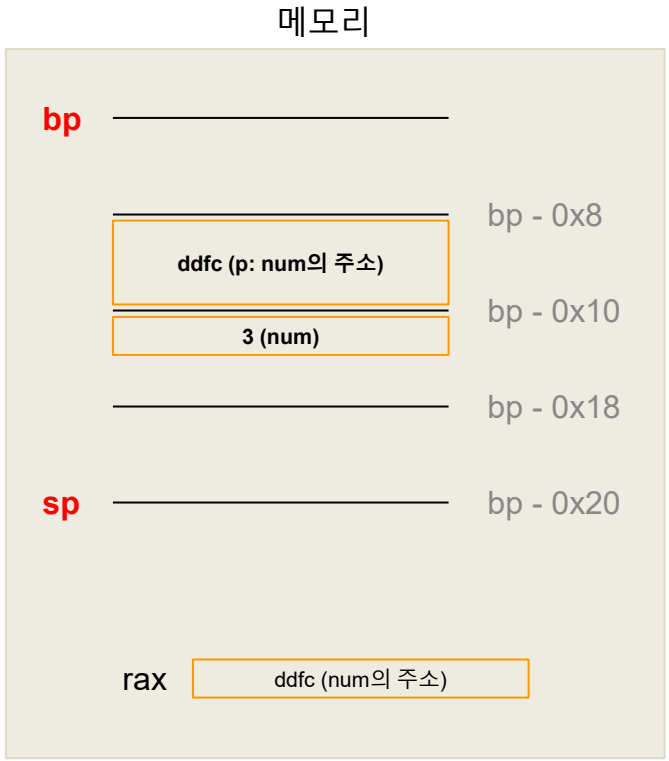
- p 명령어를 사용하여 변수의 주소를 알아낼 수 있다.

```
(gdb) p &num
$9 = (int *) 0x7fffffffddfc     ← num의 주소
(gdb) p/x $rax
$10 = 0x7fffffffddfc            ← rax 값 == num의 주소
```

lea 명령어 실행 직후, rax에 저장된 값 확인

포인터 변수 선언과 참조의 어셈블리어 분석

3-3) 스택에 배치된 변수를 그림으로 그려보자



설명

- 포인터 변수를 스택에 배치하는 방법
  - lea 명령어로 스택에 배치된 num의 주소를 rax에 저장함
  - 그 다음, mov 명령어로 rax 값을 스택에 배치함

포인터 변수 선언과 참조의 어셈블리어 분석

EDDI
Electronic Design Development Institute

3-4) 포인터 변수에 저장된 주소 값을 어떻게 참조하는지 분석

```
(gdb) disas
Dump of assembler code for function main:
=> 0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   %rbp
   0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x20,%rsp
   0x0000555555555175 <+12>:    mov    %fs:0x28,%rax
   0x000055555555517e <+21>:    mov    %rax,-0x8(%rbp)
   0x0000555555555182 <+25>:    xor    %eax,%eax
   0x0000555555555184 <+27>:    movl   $0x3,-0x14(%rbp)
   0x000055555555518b <+34>:    lea    -0x14(%rbp),%rax
   0x000055555555518f <+38>:    mov    %rax,-0x10(%rbp)
   0x0000555555555193 <+42>:    mov    -0x10(%rbp),%rax
   0x0000555555555197 <+46>:    movl   $0x7,(%rax)
   0x000055555555519d <+52>:    mov    -0x14(%rbp),%eax
   0x00005555555551a0 <+55>:    mov    %eax,%esi
   0x00005555555551a2 <+57>:    lea    0xe5b(%rip),%rdi        # 0x555555556004
   0x00005555555551a9 <+64>:    mov    $0x0,%eax
   0x00005555555551ae <+69>:    callq  0x555555555070 <printf@plt>
```

← 스택에 배치된 포인터 변수 p 값을 rax로 복사,
7을 rax에 저장된 주소값의 위치(변수 num)에 저장
(*p = 7;)

num의 값이 3에서 7로 바뀜

← printf("num = %d\n", num);

포인터 변수 선언과 참조의 어셈블리어 분석

4) 정리

포인터 변수를 스택에 배치하는 방법

1. (lea 명령어) 스택에 배치된 변수의 메모리 주소를 rax로 저장한다.
2. (mov 명령어) rax 값을 스택에 배치한다.

*** 연산자로 포인터 변수를 참조하는 방법**

1. (mov 명령어) 스택에 배치된 포인터 변수 값을 rax로 복사한다.
2. (mov 명령어) (%rax) 괄호를 사용하여 저장할 값을 포인터를 참조하여 복사한다.

5) 결론

포인터 변수도 메모리다.

포인터 변수 선언과 참조의 어셈블리어 분석

EDDI
Electronic Design Development Institute

6) 참고

이중 포인터는 왜 써야하는가

- 재귀호출을 사용하지 않고 리스트를 구현하려면 단일 포인터로 충분하다.
- 하지만, 트리의 재귀호출을 없애려면 단일 포인터가 아닌 이중 포인터를 사용해야 한다.

단일 포인터 vs 이중 포인터

- 단일 포인터는 속도 측면에서 좋음

사실 학교에서 자료구조 과목을 수강하지 않아서, 잘 이해가 되지 않지만 이 내용은 Lv2에서 다룬다고 한다.
(Lv1 끝나면 바로 Lv2 수강할 생각이다)

포인터 배열 어셈블리어 분석

EDDI
Electronic Design Development Institute

1) ptr_arr.c (포인터 배열 = 포인터의 배열)

```c
#include <stdio.h>

int main(void)
{
    int num1 = 3, num2 = 7, num3 = 33;
    int* arr_pointer[] = { &num1, &num2, &num3 };
    int i;

    for (i = 0; i < 3; i++)
    {
        // 우선순위: [] > *
        printf("*arr_pointer[%d] = %d\n", i, *arr_pointer[i]);
    }

    return 0;
}
```

포인터 배열 어셈블리어 분석

2) ptr_arr.c의 어셈블리어

```
(gdb) disas
Dump of assembler code for function main:
=> 0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   %rbp
   0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x30,%rsp
   0x0000555555555175 <+12>:    mov    %fs:0x28,%rax
   0x000055555555517e <+21>:    mov    %rax,-0x8(%rbp)
   0x0000555555555182 <+25>:    xor    %eax,%eax
   0x0000555555555184 <+27>:    movl   $0x3,-0x30(%rbp)
   0x000055555555518b <+34>:    movl   $0x7,-0x2c(%rbp)
   0x0000555555555192 <+41>:    movl   $0x21,-0x28(%rbp)
   0x0000555555555199 <+48>:    lea    -0x30(%rbp),%rax
   0x000055555555519d <+52>:    mov    %rax,-0x20(%rbp)
   0x00005555555551a1 <+56>:    lea    -0x2c(%rbp),%rax
   0x00005555555551a5 <+60>:    mov    %rax,-0x18(%rbp)
   0x00005555555551a9 <+64>:    lea    -0x28(%rbp),%rax
   0x00005555555551ad <+68>:    mov    %rax,-0x10(%rbp)
   0x00005555555551b1 <+72>:    movl   $0x0,-0x24(%rbp)
   0x00005555555551b8 <+79>:    jmp    0x5555555551e0 <main+119>
   0x00005555555551ba <+81>:    mov    -0x24(%rbp),%eax
   0x00005555555551bd <+84>:    cltq
   0x00005555555551bf <+86>:    mov    -0x20(%rbp,%rax,8),%rax
   0x00005555555551c4 <+91>:    mov    (%rax),%edx
   0x00005555555551c6 <+93>:    mov    -0x24(%rbp),%eax
   0x00005555555551c9 <+96>:    mov    %eax,%esi
   0x00005555555551cb <+98>:    lea    0xe32(%rip),%rdi        # 0x555555556004
   0x00005555555551d2 <+105>:   mov    $0x0,%eax
   0x00005555555551d7 <+110>:   callq  0x555555555070 <printf@plt>
   0x00005555555551dc <+115>:   addl   $0x1,-0x24(%rbp)
   0x00005555555551e0 <+119>:   cmpl   $0x2,-0x24(%rbp)
   0x00005555555551e4 <+123>:   jle    0x5555555551ba <main+81>
   0x00005555555551e6 <+125>:   mov    $0x0,%eax
   0x00005555555551eb <+130>:   mov    -0x8(%rbp),%rcx
   0x00005555555551ef <+134>:   xor    %fs:0x28,%rcx
   0x00005555555551f8 <+143>:   je     0x5555555551ff <main+150>
   0x00005555555551fa <+145>:   callq  0x555555555060 <__stack_chk_fail@plt>
   0x00005555555551ff <+150>:   leaveq
   0x0000555555555200 <+151>:   retq
End of assembler dump.
```

포인터 배열 어셈블리어 분석

EDDI
Electronic Design Development Institute

3-1) 변수 할당 간단하게 분석

```
(gdb) disas
Dump of assembler code for function main:
=> 0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   %rbp
   0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x30,%rsp
   0x0000555555555175 <+12>:    mov    %fs:0x28,%rax
   0x000055555555517e <+21>:    mov    %rax,-0x8(%rbp)
   0x0000555555555182 <+25>:    xor    %eax,%eax
   0x0000555555555184 <+27>:    movl   $0x3,-0x30(%rbp)
   0x000055555555518b <+34>:    movl   $0x7,-0x2c(%rbp)
   0x0000555555555192 <+41>:    movl   $0x21,-0x28(%rbp)
   0x0000555555555199 <+48>:    lea    -0x30(%rbp),%rax
   0x000055555555519d <+52>:    mov    %rax,-0x20(%rbp)
   0x00005555555551a1 <+56>:    lea    -0x2c(%rbp),%rax
   0x00005555555551a5 <+60>:    mov    %rax,-0x18(%rbp)
   0x00005555555551a9 <+64>:    lea    -0x28(%rbp),%rax
   0x00005555555551ad <+68>:    mov    %rax,-0x10(%rbp)
   0x00005555555551b1 <+72>:    movl   $0x0,-0x24(%rbp)
   0x00005555555551b8 <+79>:    jmp    0x5555555551e0 <main+119>
```

← 48바이트 스택 생성

← int형 변수 할당

← int*형 배열 할당 (int* arr_pointer[] = …;)

← 변수 i 할당 (for의 초기식)

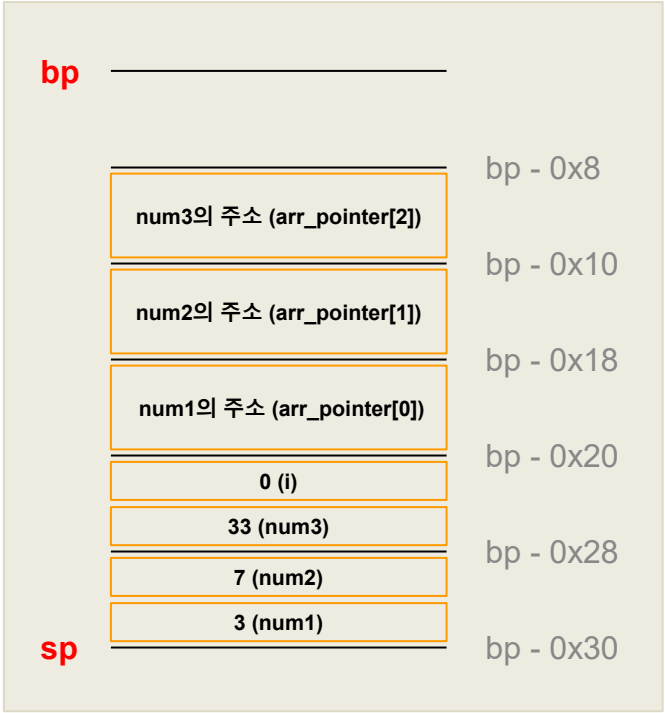
설명

int형 변수 4개, int*형 변수 3개이므로 총 40바이트이다.

포인터 배열 어셈블리어 분석

3-2) 스택에 배치된 변수를 그림으로 그려보자

메모리



설명

- int*형 배열이 스택에 배치되었는데, 어셈블리어를 보면 int*형 변수를 여러 개 선언하여 배치한 것과 다름이 없다.
- 포인터 배열 또한 메모리이다.

포인터 배열 어셈블리어 분석

EDDI
Electronic Design Development Institute

3-3) 포인터 배열 요소에 어떻게 참조하는지 분석

```
   0x00005555555551b8 <+79>:    jmp    0x5555555551e0 <main+119>
   0x00005555555551ba <+81>:    mov    -0x24(%rbp),%eax
   0x00005555555551bd <+84>:    cltq
   0x00005555555551bf <+86>:    mov    -0x20(%rbp,%rax,8),%rax
   0x00005555555551c4 <+91>:    mov    (%rax),%edx
   0x00005555555551c6 <+93>:    mov    -0x24(%rbp),%eax
   0x00005555555551c9 <+96>:    mov    %eax,%esi
   0x00005555555551cb <+98>:    lea    0xe32(%rip),%rdi        # 0x555555556004
   0x00005555555551d2 <+105>:   mov    $0x0,%eax
   0x00005555555551d7 <+110>:   callq  0x555555555070 <printf@plt>
   0x00005555555551dc <+115>:   addl   $0x1,-0x24(%rbp)
   0x00005555555551e0 <+119>:   cmpl   $0x2,-0x24(%rbp)
   0x00005555555551e4 <+123>:   jle    0x5555555551ba <main+81>
   0x00005555555551e6 <+125>:   mov    $0x0,%eax
   0x00005555555551eb <+130>:   mov    -0x8(%rbp),%rcx
   0x00005555555551ef <+134>:   xor    %fs:0x28,%rcx
   0x00005555555551f8 <+143>:   je     0x5555555551ff <main+150>
   0x00005555555551fa <+145>:   callq  0x555555555060 <__stack_chk_fail@plt>
   0x00005555555551ff <+150>:   leaveq
   0x0000555555555200 <+151>:   retq
End of assembler dump.
```

← 배열 요소를 edx에 복사 (자세한 내용은 뒤에서...)

포인터 배열 어셈블리어 분석

EDDI
Electronic Design Development Institute

3-4) 포인터 배열 요소에 어떻게 참조하는지 분석

int*형 배열의 요소에 접근하는 방법은 '배열 선언의 어셈블리어 분석 *(array.c)*'에서 설명한 것과 **똑같다**.

```
0x00005555555551b8 <+79>:    jmp    0x5555555551e0 <main+119>
0x00005555555551ba <+81>:    mov    -0x24(%rbp),%eax
0x00005555555551bd <+84>:    cltq
0x00005555555551bf <+86>:    mov    -0x20(%rbp,%rax,8),%rax
0x00005555555551c4 <+91>:    mov    (%rax),%edx
0x00005555555551c6 <+93>:    mov    -0x24(%rbp),%eax
0x00005555555551c9 <+96>:    mov    %eax,%esi
0x00005555555551cb <+98>:    lea    0xe32(%rip),%rdi        # 0x555555556004
0x00005555555551d2 <+105>:   mov    $0x0,%eax
0x00005555555551d7 <+110>:   callq  0x5555555555070 <printf@plt>
```

printf 함수의 세 번째 인자로 *arr_pointer[i] 값을 넘겨주는 방법 분석

- mov -0x20(%rbp, %rax, 8), %rax
  - mov 명령어로 배열의 요소를 rax에 저장한다.
  - 여기서 8은 배열 요소의 자료형(int*) 크기이다.

- mov (%rax), %edx
  - rax 값은 int*이므로 괄호를 사용(포인터 참조)하여 edx에 저장하고 있다.

포인터 배열 어셈블리어 분석

EDDI
Electronic Design Development Institute

4) 결론

포인터 배열 선언은 포인터 변수를 여러 개 선언한 것과 다름 없다. 이 또한 메모리이다.

5) 정리

C언어에서 배열을 사용하여 선언하는 것을 어셈블리어로 보면 배열의 요소 하나하나를 배치한 것인데 이것은 변수 하나를 여러 개 선언한 것과 다르지 않다.
(스택에 변수를 배치하는 명령어만 보고 C 레벨에서 배열로 선언했는지 변수로 선언했는지 구분하라고 하면 못할듯)

그러므로 C언어에서는 어셈블리어 보다 일괄적인 관리를 위해 배열의 문법을 추가함으로써 추상화했다고 볼 수 있다.

6) 참고

- * 앞에 데이터 타입이 있으면 참조 연산이 아님 **(데이터 타입임)**
  - (int *) p
- * 앞에 데이터 타입이 없으면 참조 연산
  - *p

배열 포인터 어셈블리어 분석

EDDI
Electronic Design Development Institute

1) arr_ptr.c

```c
#include <stdio.h>

int main(void)
{
    int i;
    int arr[3][3] = {
        { 1, 2, 3 },
        { 4, 5, 6 },
        { 7, 8, 9 },
    };

    // 배열 포인터
    int (*p)[4] = arr;

    for (i = 0; i < 3; i++)
    {
        printf("*p[%d] = %d\n", i, *p[i]);
    }

    return 0;
}
```

배열 포인터 어셈블리어 분석

2) arr_ptr.c 컴파일 시 경고 발생

```
try@try-desktop:~/Desktop/eddi/5_week$ gcc -g -o arr_ptr arr_ptr.c
arr_ptr.c: In function ‘main’:
arr_ptr.c:13:16: warning: initialization of ‘int (*)[4]’ from incompatible pointer type ‘int (*)[3]’
[-Wincompatible-pointer-types]
   13 |  int (*p)[4] = arr;
      |                ^~~
```

```
try@try-desktop:~/Desktop/eddi/5_week$ ./arr_ptr
*p[0] = 1
*p[1] = 5
*p[2] = 9
```

실행 결과

설명

- 컴파일 시 데이터타입 경고가 발생하지만, 실행은 잘 된다.

배열 포인터 어셈블리어 분석

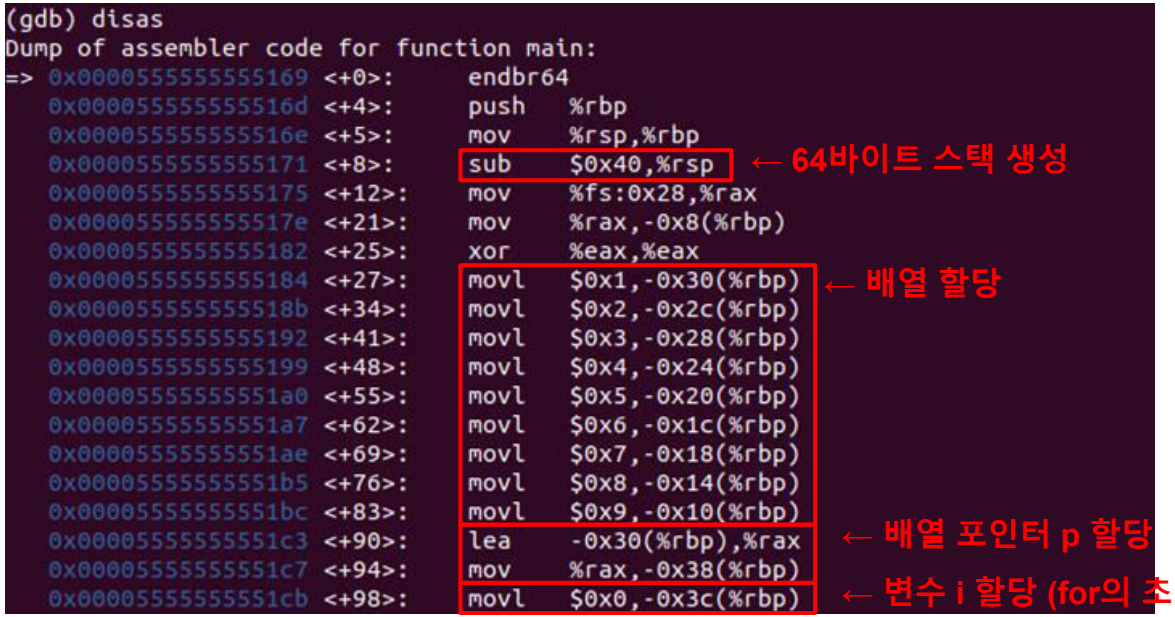
EDDI
Electronic Design Development Institute

3) arr_ptr.c의 어셈블리어

```
(gdb) disas
Dump of assembler code for function main:
=> 0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   %rbp
   0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x40,%rsp
   0x0000555555555175 <+12>:    mov    %fs:0x28,%rax
   0x000055555555517e <+21>:    mov    %rax,-0x8(%rbp)
   0x0000555555555182 <+25>:    xor    %eax,%eax
   0x0000555555555184 <+27>:    movl   $0x1,-0x30(%rbp)
   0x000055555555518b <+34>:    movl   $0x2,-0x2c(%rbp)
   0x0000555555555192 <+41>:    movl   $0x3,-0x28(%rbp)
   0x0000555555555199 <+48>:    movl   $0x4,-0x24(%rbp)
   0x00005555555551a0 <+55>:    movl   $0x5,-0x20(%rbp)
   0x00005555555551a7 <+62>:    movl   $0x6,-0x1c(%rbp)
   0x00005555555551ae <+69>:    movl   $0x7,-0x18(%rbp)
   0x00005555555551b5 <+76>:    movl   $0x8,-0x14(%rbp)
   0x00005555555551bc <+83>:    movl   $0x9,-0x10(%rbp)
   0x00005555555551c3 <+90>:    lea    -0x30(%rbp),%rax
   0x00005555555551c7 <+94>:    mov    %rax,-0x38(%rbp)
   0x00005555555551cb <+98>:    movl   $0x0,-0x3c(%rbp)
   0x00005555555551d2 <+105>:   jmp    0x555555555203 <main+154>
   0x00005555555551d4 <+107>:   mov    -0x3c(%rbp),%eax
   0x00005555555551d7 <+110>:   cltq
   0x00005555555551d9 <+112>:   shl    $0x4,%rax
   0x00005555555551dd <+116>:   mov    %rax,%rdx
   0x00005555555551e0 <+119>:   mov    -0x38(%rbp),%rax
   0x00005555555551e4 <+123>:   add    %rdx,%rax
   0x00005555555551e7 <+126>:   mov    (%rax),%edx
   0x00005555555551e9 <+128>:   mov    -0x3c(%rbp),%eax
   0x00005555555551ec <+131>:   mov    %eax,%esi
   0x00005555555551ee <+133>:   lea    0xe0f(%rip),%rdi        # 0x555555556004
   0x00005555555551f5 <+140>:   mov    $0x0,%eax
   0x00005555555551fa <+145>:   callq  0x555555555070 <printf@plt>
   0x00005555555551ff <+150>:   addl   $0x1,-0x3c(%rbp)
   0x0000555555555203 <+154>:   cmpl   $0x2,-0x3c(%rbp)
   0x0000555555555207 <+158>:   jle    0x5555555551d4 <main+107>
   0x0000555555555209 <+160>:   mov    $0x0,%eax
   0x000055555555520e <+165>:   mov    -0x8(%rbp),%rcx
   0x0000555555555212 <+169>:   xor    %fs:0x28,%rcx
   0x000055555555521b <+178>:   je     0x555555555222 <main+185>
   0x000055555555521d <+180>:   callq  0x555555555060 <__stack_chk_fail@plt>
   0x0000555555555222 <+185>:   leaveq
   0x0000555555555223 <+186>:   retq
End of assembler dump.
```

배열 포인터 어셈블리어 분석

4-1) arr_ptr.c의 어셈블리어 중요 포인트 분석



(참고)

- 2차원 배열이지만 실제 RAM은 1차원 구조이기 때문에 순서대로 배치한다.
- C 소스코드에서 변수 i를 맨 첫 번째로 선언했지만, 어셈블리어에서는 초기화를 해야 스택에 배치되므로 i가 맨 마지막에 할당되었다.

배열 포인터 어셈블리어 분석

4-2) 스택에 배치된 변수를 그림으로 그려보자

배열 포인터 어셈블리어 분석

EDDI
Electronic Design Development Institute

4-3) 배열 요소에 접근하는 방법

```
0x00005555555551d2 <+105>:    jmp    0x555555555203 <main+154>
0x00005555555551d4 <+107>:    mov    -0x3c(%rbp),%eax
0x00005555555551d7 <+110>:    cltq
0x00005555555551d9 <+112>:    shl    $0x4,%rax
0x00005555555551dd <+116>:    mov    %rax,%rdx
0x00005555555551e0 <+119>:    mov    -0x38(%rbp),%rax
0x00005555555551e4 <+123>:    add    %rdx,%rax
0x00005555555551e7 <+126>:    mov    (%rax),%edx
0x00005555555551e9 <+128>:    mov    -0x3c(%rbp),%eax
0x00005555555551ec <+131>:    mov    %eax,%esi
0x00005555555551ee <+133>:    lea    0xe0f(%rip),%rdi        # 0x555555556004
0x00005555555551f5 <+140>:    mov    $0x0,%eax
0x00005555555551fa <+145>:    callq  0x555555555070 <printf@plt>
0x00005555555551ff <+150>:    addl   $0x1,-0x3c(%rbp)
0x0000555555555203 <+154>:    cmpl   $0x2,-0x3c(%rbp)
0x0000555555555207 <+158>:    jle    0x5555555551d4 <main+107>
0x0000555555555209 <+160>:    mov    $0x0,%eax
0x000055555555520e <+165>:    mov    -0x8(%rbp),%rcx
0x0000555555555212 <+169>:    xor    %fs:0x28,%rcx
0x000055555555521b <+178>:    je     0x555555555222 <main+185>
0x000055555555521d <+180>:    callq  0x555555555060 <__stack_chk_fail@plt>
0x0000555555555222 <+185>:    leaveq
0x0000555555555223 <+186>:    retq
```

(참고)

- 4칸씩(16byte) 건너뛰는 방법으로 왼쪽 시프트 연산을 하는 shl 명령어를 사용했다.

배열 포인터 어셈블리어 분석

EDDI
Electronic Design Development Institute

5. 파일이름: arr_ptr.c

int **p[3]** → **int (***)[3] 4byte 포인터 배열
int (*p)[4] → int[4] * 16byte 배열 포인터

수업 시간에 들었지만, 배열 포인터 타입이 정확히 이해가 안된다...

(추가 학습이 필요하다)

AND NOT 기법

EDDI
Electronic Design Development Institute

4096~8191 중 숫자 하나와 AND NOT(2^4 - 1) 연산하는 경우

12	11	10	9	8	7	6	5	4	3	2	1	0	2의 지수
1	1	1	1	1	1	1	1	1	0	0	0	0	NOT(2^4 - 1)
x	x	x	x	x	x	x	x	x	x	x	x	x	AND 4096 ~ 8191

x	x	x	x	x	x	x	x	x	0	0	0	0

2^{12} 2^{11} 2^{10} 2^{9} 2^{8} 2^{7} 2^{6} 2^{5} 2^{4}

∴ 2의 4제곱의 배수만 남기때문에, 비트 연산 2번 만에 가장 가까운 16의 배수를 구했다.

2차원 배열을 반환하는 함수

EDDI
Electronic Design Development Institute

1) private_arr_return.c

```c
#include <stdio.h>

int (* arr_return_test(void))[2]
{
    static int arr[2][2] = {
        { 1, 2 },
        { 3, 4 }
    };

    return arr;
}

int main(void)
{
    int (*p)[2] = arr_return_test();
    int i, j;

    for (i = 0; i < 2; i++)
    {
        printf("*p[%d] = %d\n", i, *p[i]);
    }

    //arr[0][0] = 7;

    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 2; j++)
        {
            printf("p[%d][%d] = %d\n", i, j, p[i][j]);
        }
    }

    return 0;
}
```

2차원 배열을 반환하는 함수

EDDI
Electronic Design
Development Institute

2) 실행 결과

```
try@try-desktop:~/Desktop/eddi/5_week$ ./private_arr_return
*p[0] = 1
*p[1] = 3
p[0][0] = 1
p[0][1] = 2
p[1][0] = 3
p[1][1] = 4
```

2차원 배열을 반환하는 함수

3) 2차원 배열을 반환하는 함수 프로토타입

```
int (* arr_return_test(void))[2];
```

carbon
carbon.now.sh

프로토타입 분석

- 반환형: int (*)[2]
- 함수 이름: arr_return_test
- 매개변수: (void)

일반적으로 함수를 정의하는 방법은 '반환형 함수이름(매개변수)' 형태인데, 만약 이 형태로 함수 arr_return_test를 선언했다면 **'int (*)[2] arr_return_test(void)'** 이렇게 작성할 수 있다.

하지만, 위와 같은 형태로는 선언할 수 없기 때문에 반환형의 일부를 뒤쪽으로 빼는 형태로만 선언할 수 있다고 생각하면 된다.

그러므로 배열이나 함수 포인터를 반환하는 함수를 작성할 경우에는 다음과 같이 작성하면 된다.

- **int (*)[2]** arr_return_test(void) → **int (*** arr_return_test(void))**[2]** *(이해하기 쉽게 반환형을 빨간색으로 칠함)*

2차원 배열을 반환하는 함수

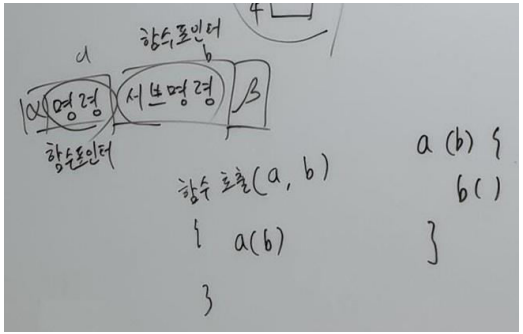
4) (참고) 함수 포인터 연습

① 매개변수로 함수 포인터를 받고, 함수 포인터를 반환하는 함수를 작성한다고 하면 다음과 같다.

```
int (* return_function_pointer(int (*fp)(void)))(void)
```

② 매개변수로 메인 명령어와 서브 명령어를 함수 포인터로 받고 내부에서 실행하는 함수를 작성한다면 다음과 같다.

```
void protocol(void (*main_command)(void (*)(void)), void (*sub_command)(void))
```



왼쪽 그림을 보면 a라는 함수에 서브 명령어인 b를 인자로 넣고 있기 때문에 a는 함수 포인터를 매개변수로 선언해야 한다.

근데 내가 작성한 것이 정확한지 모르겠다...

2차원 배열을 반환하는 함수

5) 정리

맨 처음에 **int (* arr_return_test(void))[2]**’가 함수라는 것을 보고 멘탈이 나갔다.

하지만 곰곰히 생각을 해보니 반환형에는 이름이 없기 때문에 어쩔 수 없이 이렇게 작성하는 거라고 기억하면 편하다.

2차원 배열을 반환하는 함수 (문서)

EDDI
Electronic Design Development Institute

1) 구글에 ‘return 2d array pointer type in c’ 검색

2) https://stackoverflow.com/questions/69043369/returning-2d-array-in-c

함수 포인터를 반환하는 함수 (문서)

EDDI
Electronic Design Development Institute

1) https://stackoverflow.com/questions/20617067/returning-function-pointer-type

2) https://stackoverflow.com/questions/34565532/is-it-possible-to-return-a-function-from-another-function-in-c?noredirect=1&lq=1

3) https://dojang.io/mod/page/view.php?id=600

4) https://welikecse.tistory.com/56

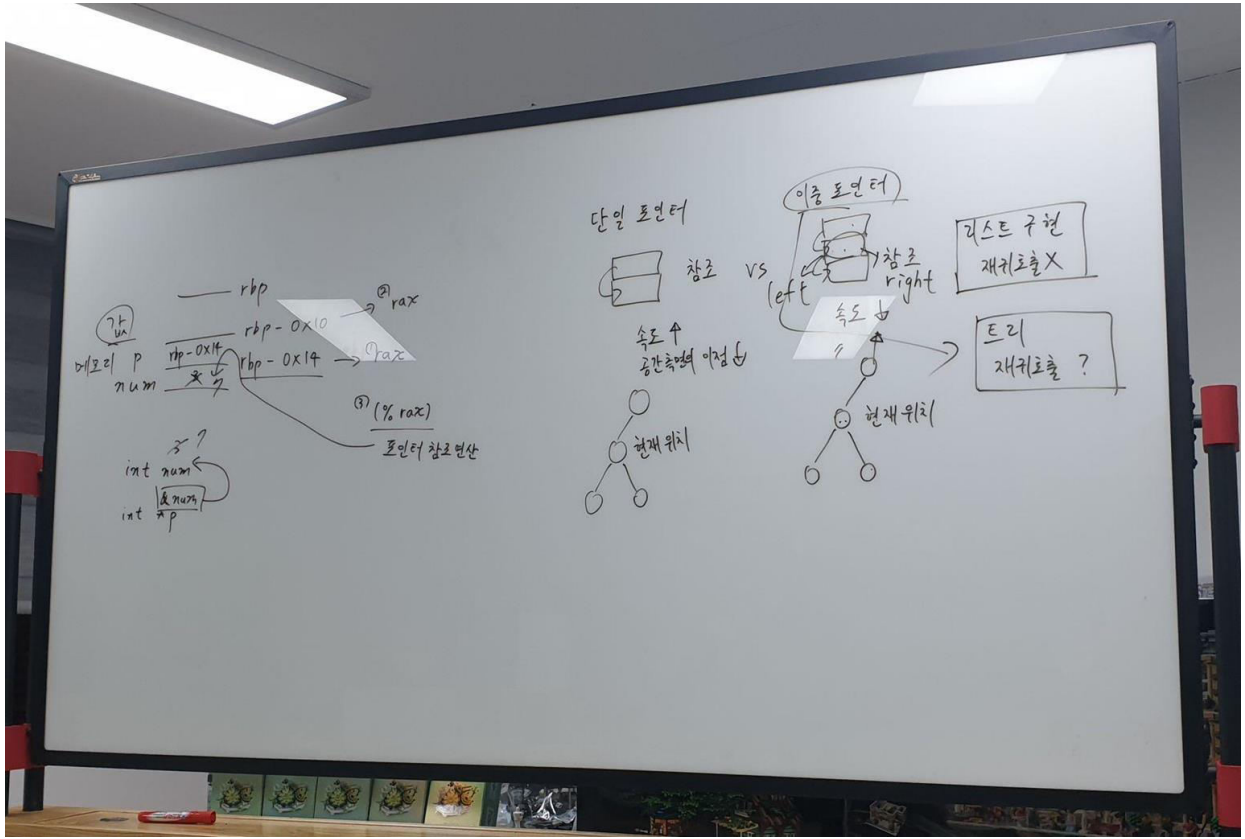
5) 구글에 ‘void (*signal(int sig, void (*handler) (int))) (int);’ 검색

5-2) https://stackoverflow.com/questions/3706704/whats-the-meaning-of-this-piece-of-code-void-signalint-sig-void-funcin

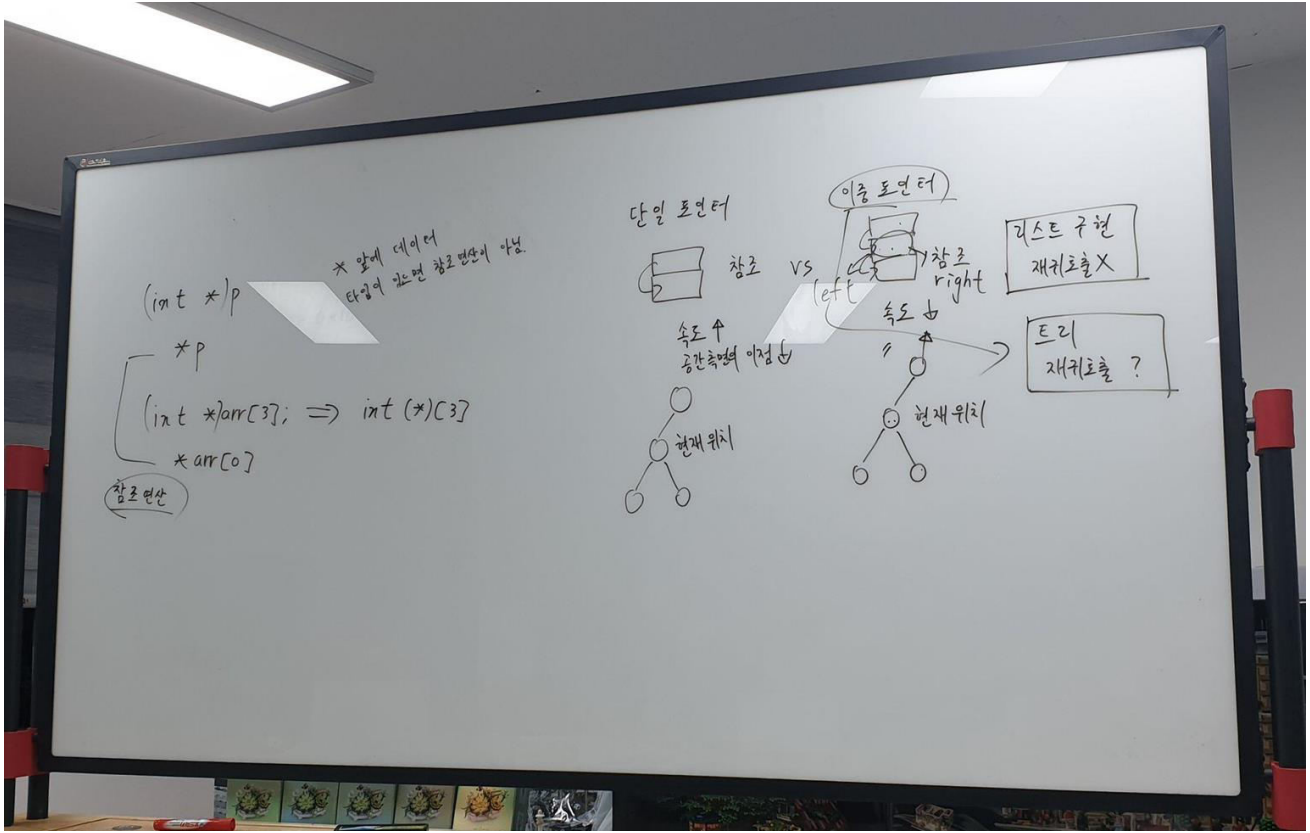
5-3) https://stackoverflow.com/questions/9500848/how-do-i-read-this-complex-declaration-in-c/9501054#9501054

5-4) https://kldp.org/node/43211

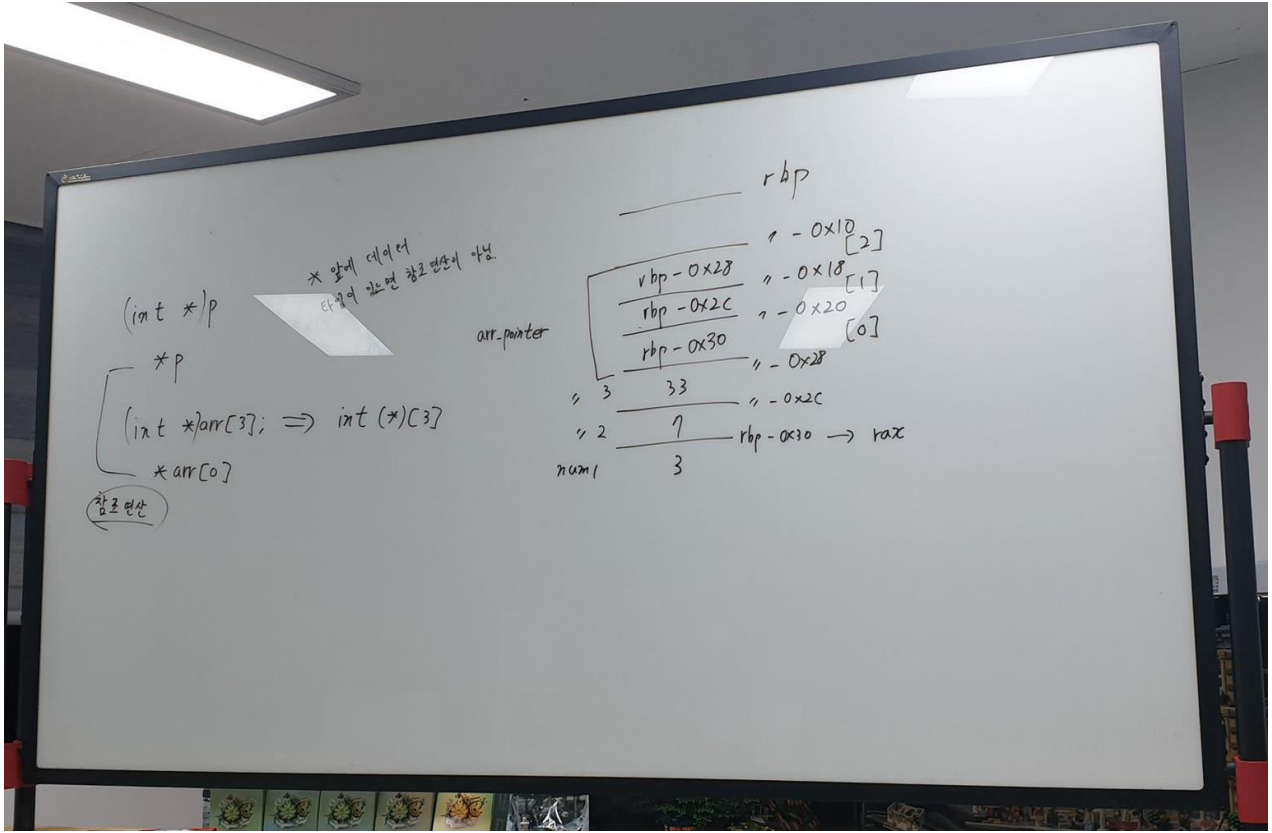
수업내용 사진


rbp
rbp - 0x10
rbp - 0x14
메모리
num
int num
int *p
(% rax)
포인터 참조연산
단일 포인터
참조
vs
이중 포인터
left
right
속도
공간측면의 이점
현재위치
현재위치
리스트 구현
재귀호출X
트리
재귀호출 ?

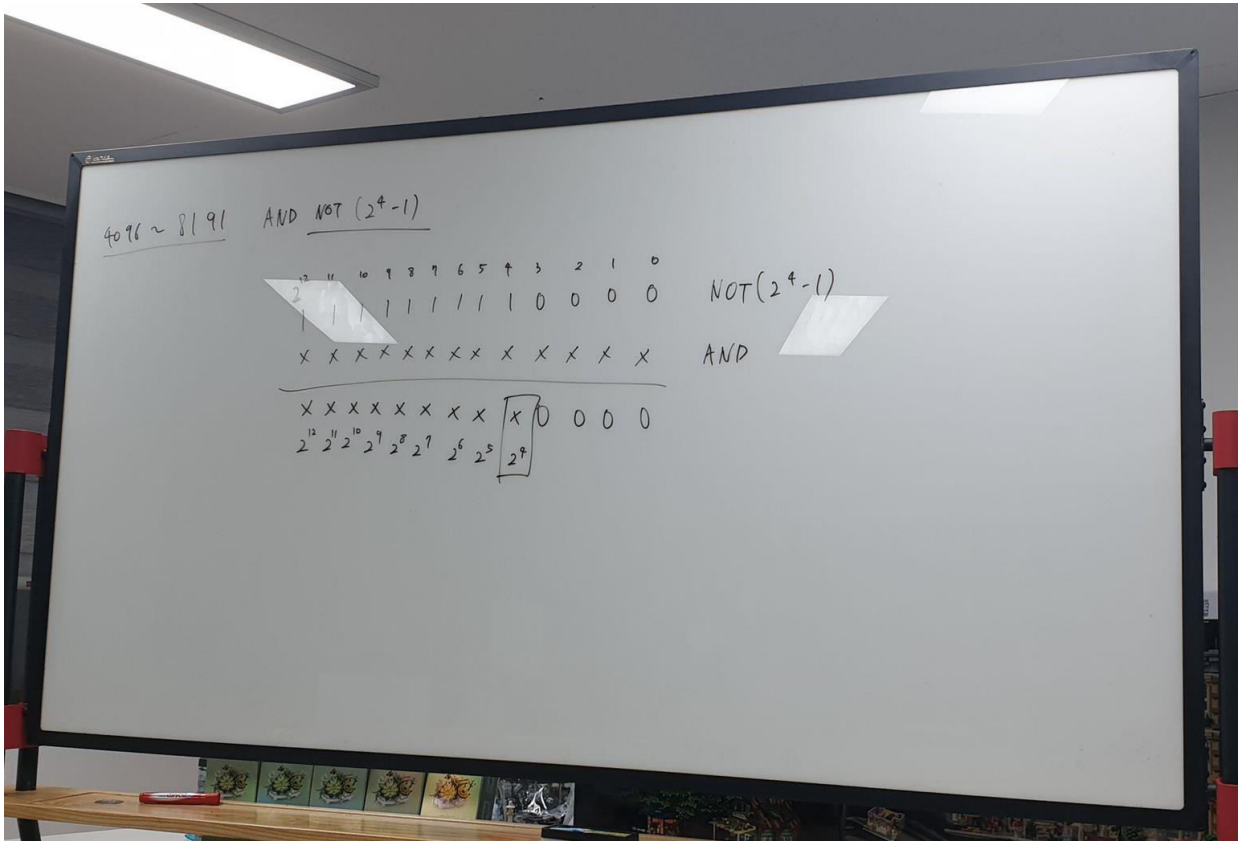
수업내용 사진


(int *)p
* 앞에 데이터 타입이 있으면 참조연산이 아님.
*p
(int *)arr[3]; ⇒ int (*)[3]
*arr[0]
참조연산
단일 포인터
참조
vs
이중 포인터
참조
right
left
속도 ↑
공간측면 이점 ↓
현재위치
속도 ↓
현재위치
리스트 구현
재귀호출 X
트리
재귀호출 ?

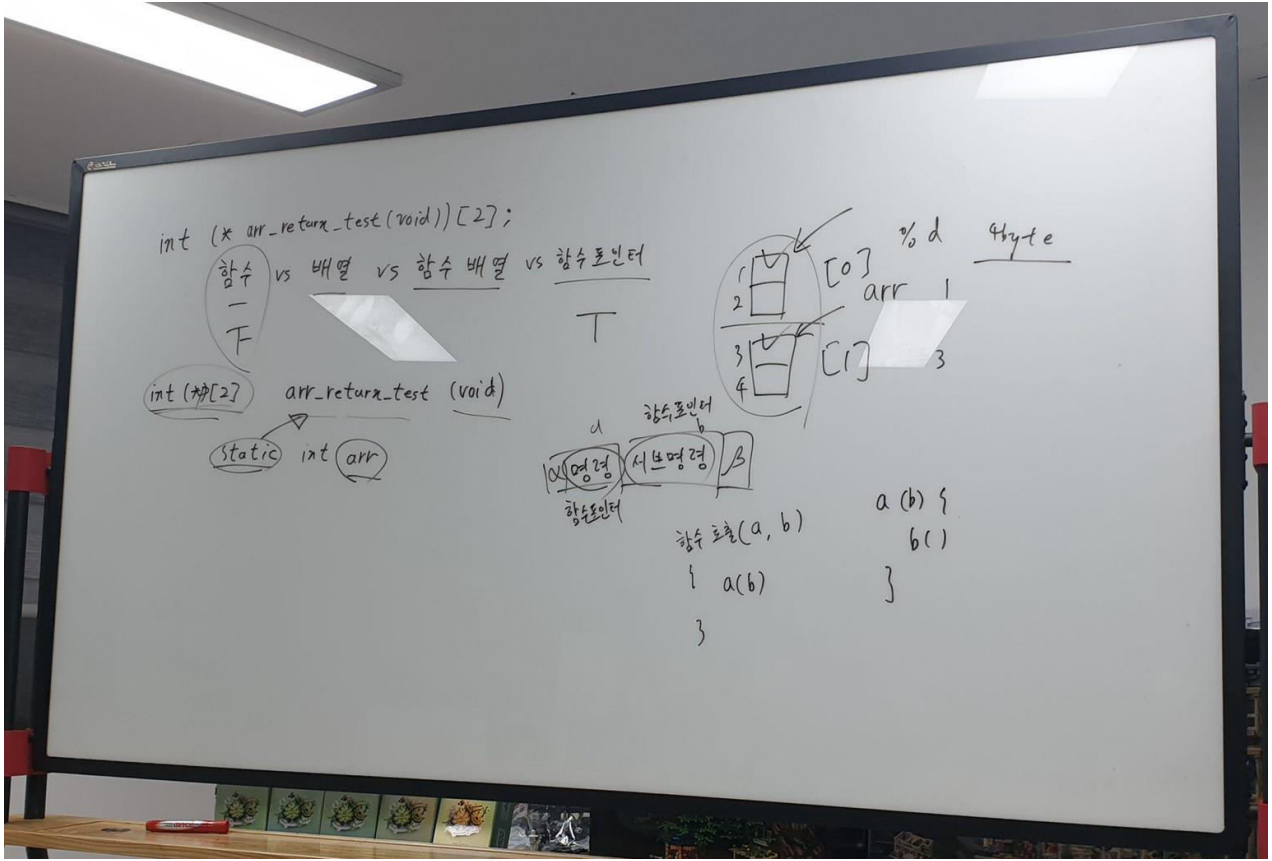
수업내용 사진


rbp
- 0x10 [2]
- 0x18 [1]
- 0x20 [0]
rbp - 0x28
rbp - 0x2C
rbp - 0x30
- 0x28
- 0x2C
rbp - 0x30 → rax
arr_pointer
num1
(int *)p
*p
(int *)arr[3]; ⇒ int (*)[3]
*arr[0]
참조연산
* 앞에 데이터 타입이 있으면 참조연산이 아님.

수업내용 사진



수업내용 사진


int (* arr_return_test(void))[2];
함수 vs 배열 vs 함수 배열 vs 함수포인터
int (*)[2]
arr_return_test (void)
static
int
arr
함수포인터
명령
서브명령
함수 호출(a, b)
{ a(b)
}
a(b) {
b()
}
%d
4byte
arr
[0]
[1]