# Computer Architecture


Network Computing Lab.

## Arm assembly Lab 2

# Contents

- **Lab2. GDB를 이용한 C pointer/ buffer overflow 동작 이해**
  - 2-1 : C pointer 동작 사례 1
  - 2-2 : C pointer 동작 사례 2
  - 2-3 : DT(Data Transfer) instruction에서의 indexing
  - 2-4 : C buffer overflow
  - HW6: 실습 A ~ D 수행 후 결과내용을 캡쳐 또는 내용 작성하여 보고서로 제출하세요.

# Lab 2-1 : C pointer 동작 사례 1

- C pointer의 기본 예제를 통해 pointer의 개념을 학습한다.
- C pointer는 assembly로 어떻게 표현되는가를 살펴보고 pointer 값 변화가 메모리 값으로는 어떻게 변화하는지를 이해한다.

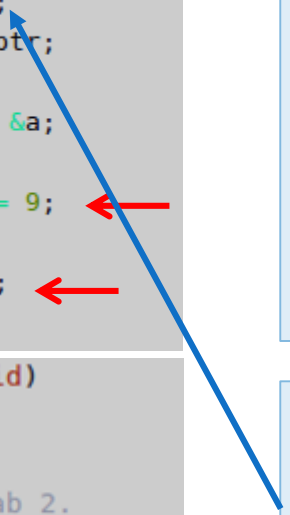
# Lab 2-1 : C pointer 동작 사례 1

## Debugging

Break point

```
19  void example1()
20  {
21          int a;
22          int *ptr;
23
24          ptr = &a;
25
26          *ptr = 9;
27
28          a = 3;
29  }
```

```
7   int main (void)
8   {
9           /*
10           * lab 2.
11           * Write down th
12           */
13
14          example1();
15
16          return 0;
17  }
```



**포인터**는 독립 변수 또는 배열 내 특정 인덱스 변수가 저장된 **메모리 주소**를 나타냅니다. ***포인터는** 변수값, 그 **메모리 주소에 저장된 값**을 나타냅니다.

왼쪽 예제에서 int형 포인터 변수인 ptr이 int형 변수 a의 메모리를 pointing하고 있습니다. (메모리주소를 가집니다)

***ptr에 9를 대입하는 과정과 a에 3을 대입하는 과정을 비교해서 봅시다. 이 때, 우리는 이 과정이 assembly에서는 어떻게 진행되는지 실습을 통해 확인합니다.**

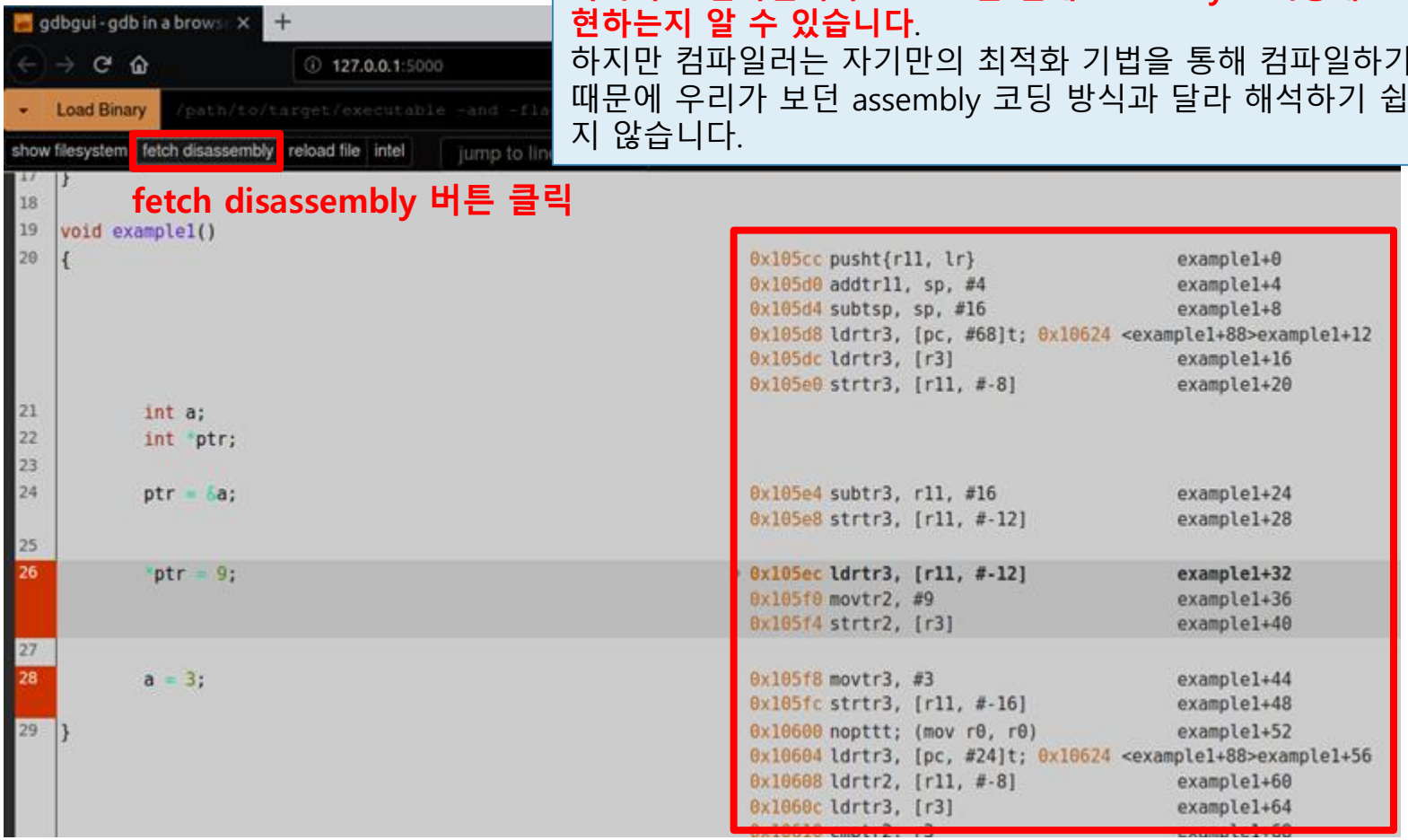
**함수 내 지역 변수는 스택 영역의 메모리를 임시로 할당하여** 함수 수행 중 사용합니다. **뒤에 예제를 따라 가보면 변수 a가 저장된 메모리에 접근할 때 sp(스택 포인터)를 이용하여 접근**하는 것을 볼 수 있습니다. (**비슷한 예제를 lab1 자료 P17에서 지역 변수 facnum, result에서 볼 수 있습니다**)

확인하고 싶은 곳을 break point를 설정한 후 Continue 버튼을 누릅니다.

# Lab 2-1 : C pointer 동작 사례 1

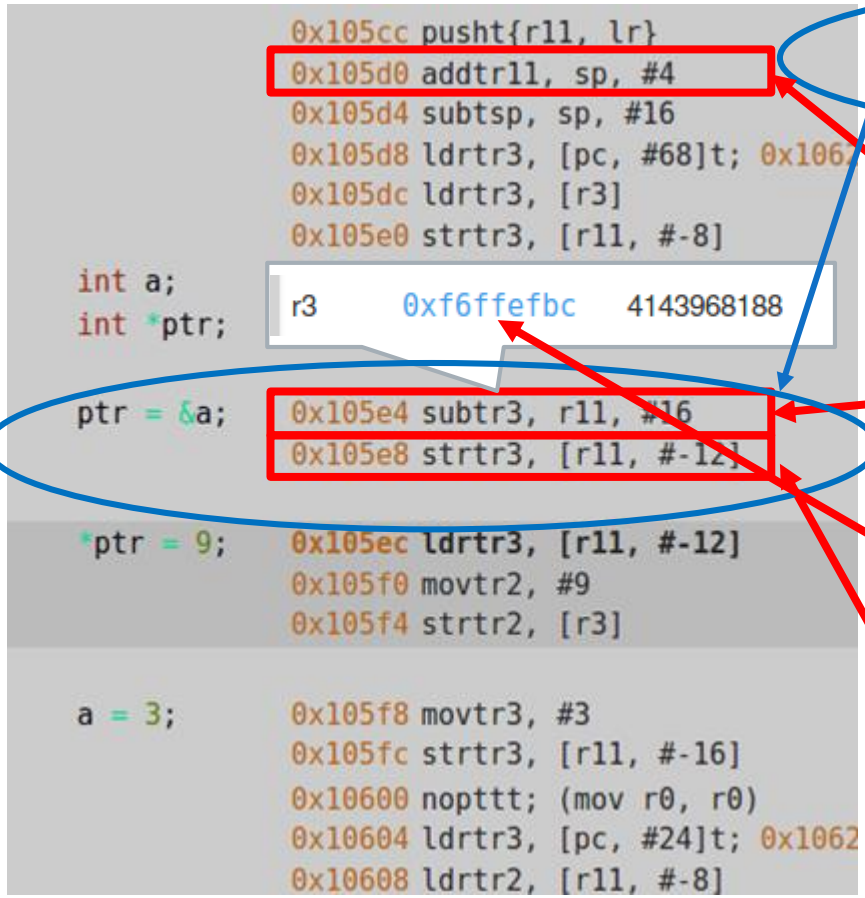
- Debugging

**fetch disassembly 버튼을 클릭하면 disassembly section이 나타나고 컴파일러가 C code를 실제 assembly로 어떻게 표현하는지 알 수 있습니다.**
하지만 컴파일러는 자기만의 최적화 기법을 통해 컴파일하기 때문에 우리가 보던 assembly 코딩 방식과 달라 해석하기 쉽지 않습니다.



# Lab 2-1 : C pointer 동작 사례 1

- Debugging




ptr = &a; 코드가 어셈블리 상으로는
sub r3, r11, #16
str r3, [r11, #-12] 이렇게 두 줄로 나왔습니다.
이해하기 쉽지 않지만 그래도 천천히 살펴보면, 위에서 r11에 sp+4를 저장하고 그 후로 sp는 업데이트된 적이 없습니다.

우리는 지역 변수는 스택에 저장한다고 배웠습니다. r11은 스택을 가리키고 있으니 아래 박스에서 **스택의 r11-16 위치에 지역 변수 a를 저장하고 바로 그 위 주소인 r11-12 위치에는 지역 변수 ptr 즉, 지역변수 a의 주소값을 저장한다고** 생각해보세요.

레지스터 섹션에서 r3를 확인해봅시다. 누가 봐도 메모리 주소인 것처럼 보이네요.

그 값을 메모리 r11-12 위치(ptr)에 대입하고 있어요.

아~ **동일한 주소를 사용하기 때문에 ptr에 값을 넣든 a에 값을 넣든 똑같구나**~ 생각하면 쉽죠?

그럼 실제로도 그런지 확인해 봅시다.

# Lab 2-1 : C pointer 동작 사례 1

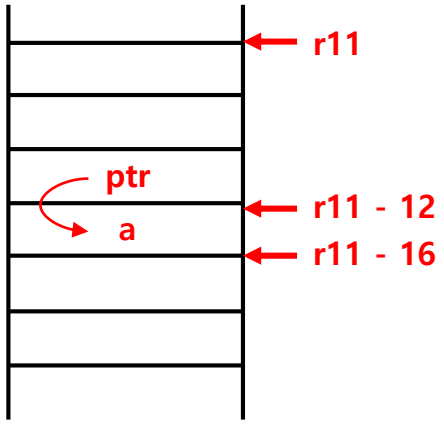
- Debugging

```
int a;
int *ptr;

ptr = &a;     0x105e4 subtr3, r11, #16
              0x105e8 strtr3, [r11, #-12]

*ptr = 9;     0x105ec ldrtr3, [r11, #-12]
              0x105f0 movtr2, #9
              0x105f4 strtr2, [r3]

a = 3;        0x105f8 movtr3, #3
              0x105fc strtr3, [r11, #-16]
              0x10600 noptt; (mov r0, r0)
              0x10604 ldrtr3, [pc, #24]t; 0
              0x10608 ldrtr2, [r11, #-8]
              0x1060c ldrtr3, [r3]
              0x10610 cmptr2, r3
```



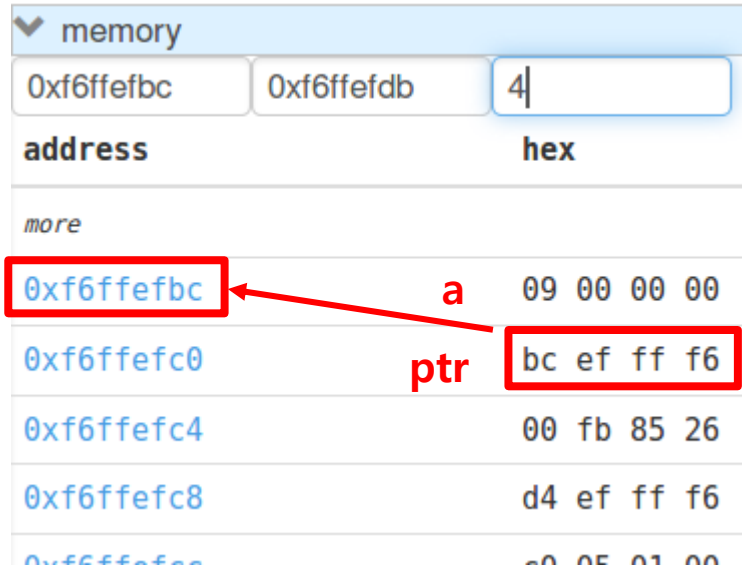

*ptr = 9; 가 어셈블리로는 다음과 같이 세 줄로 표현되었습니다.

**먼저 메모리 r11-12 번지로부터 ptr에 저장된 a의 주소값을 가져와서 r3에 저장했습니다. 그 다음 mov를 통해 r2에 상수 값 9를 대입했고(str의 준비작업) str를 통해 r2에 있는 값을 메모리 r3번지(a의 주소값)에 저장합니다.**

**레지스터 목록에서 r3에 저장된 주소값을 클릭해 메모리를 확인해봅시다.**

# Lab 2-1 : C pointer 동작 사례 1

- Debugging




**레지스터 목록에서 r3의 값을 클릭해 변수 a의 메모리를 확인해봅시다.**

일단 9가 잘 들어가 있는 것을 확인할 수 있고, 왼쪽 그림 상에서 바로 아래 주소 (실제 스택 상에서는 위에 있는) 즉, r3-12 위치(ptr)에는 a의 주소값이 저장되어 있습니다.

# Lab 2-1 : C pointer 동작 사례 1

- Debugging

```
int a;
int *ptr;

ptr = &a;       0x105e4 subtr3, r11, #16
                0x105e8 strtr3, [r11, #-12]

*ptr = 9;       0x105ec ldrtr3, [r11, #-12]
                0x105f0 movtr2, #9
                0x105f4 strtr2, [r3]

a = 3;          0x105f8 movtr3, #3
                0x105fc strtr3, [r11, #-16]
              > 0x10600 nopttt; (mov r0, r0)
                0x10604 ldrtr3, [pc, #24]t; 0x
```

memory

| 0xf6ffefbc | 0xf6ffefdb | 4 |
|---|---|---|

| address | hex |
|---|---|
| *more* | |
| 0xf6ffefbc | 03 00 00 00 |
| 0xf6ffefc0 | bc ef ff f6 |
| 0xf6ffefc4 | 00 fb 85 26 |
| 0xf6ffefc8 | d4 ef ff f6 |

전과 같이 a에 3을 대입하는 것도 비슷하게 보입니다.
이번엔 r3에 상수값 3을 대입하고 메모리 r11-16 위치에 r3를 저장했습니다.
메모리를 확인해보면 똑같은 위치에 값이 3으로 바뀐 것을 확인할 수 있습니다.

# Lab 2-1 : C pointer 동작 사례 1

- **실습 A** : p4의 example1() 함수 코드에서 아래 박스의 코드를 추가한 뒤 재빌드하세요.
- disassembly section에서 빨간색 박스 코드에 해당되는 어셈블리 코드를 확인하고 화면 캡쳐하세요.
- 디버거를 이용하여 step 별로 수행하세요.
- 빨간색 박스 코드에서 단계별 *ptr1 값과 b값을 메모리에서 확인 후 캡쳐하여 보고서에 첨부하세요.

```
int b;
int *ptr1;

ptr1 = &b;

*ptr1 = 16;

b = 256;
```

# Lab 2-2 : C pointer 동작 사례 2

- **Lab 2-1에서 포인터가 어떻게 사용되고 어떻게 메모리에 접근하는지 assembly 코드 수행 추적을 통해 공부했습니다.**
- **Lab 2-2에서는 자료형(char, int, long long)별 포인터 변수 값 변화가 메모리에서 주소를 어떻게 인덱싱하는지 살펴봅시다.**
  - char 형 포인터 변수 선언 - (예) char *a;
  - int 형 포인터 변수 선언 - (예) int *b;
  - long long 형 포인터 변수 선언 - (예) long long *c;
  - **포인터 변수 + 1 에서 +1 의미가 포인터 변수 자료형에 따라 다르다 (그 자료형 크기만큼 주소가 증가).**
    - **(예) 위 변수 선언 예제에서 a=1000 이라고 가정. d=*(a+1)을 수행하면 d에는 1001번지에 들어있는 1 byte 데이터가 저장됨. b=2000 이라고 가정. e=*(b+1)을 수행하면 e에는 2004번지부터 들어있는 4 byte 데이터가 저장됨.**

# Lab 2-2 : C pointer 동작 사례 2

Network Computing Lab.

- Debugging

```
40  void example2()
41  {
42          char mem[40] = {0, };
43          char *a;
44          int * b;
45          long long *c;
46          int i;
47
48          a = (char*)mem;
49          for (i = 0; i < 3; i++)
50          {
51                  *(a + i) = 10;
52          }
53
54          b = (int*)(mem + 4);
55          for (i = 0; i < 3; i++)
56          {
57                  *(b + i) = 1;;
58          }
59
60          c = (long long*)(mem + 16);
61          for (i = 0; i < 3; i++)
62          {
63                  *(c + i) = 12;
64          }
65  }
```

자료형별로 메모리 인덱싱을 어떻게 하는지 확인해 보는 예제입니다.

**char 1byte / int 4byte / long long 8byte**

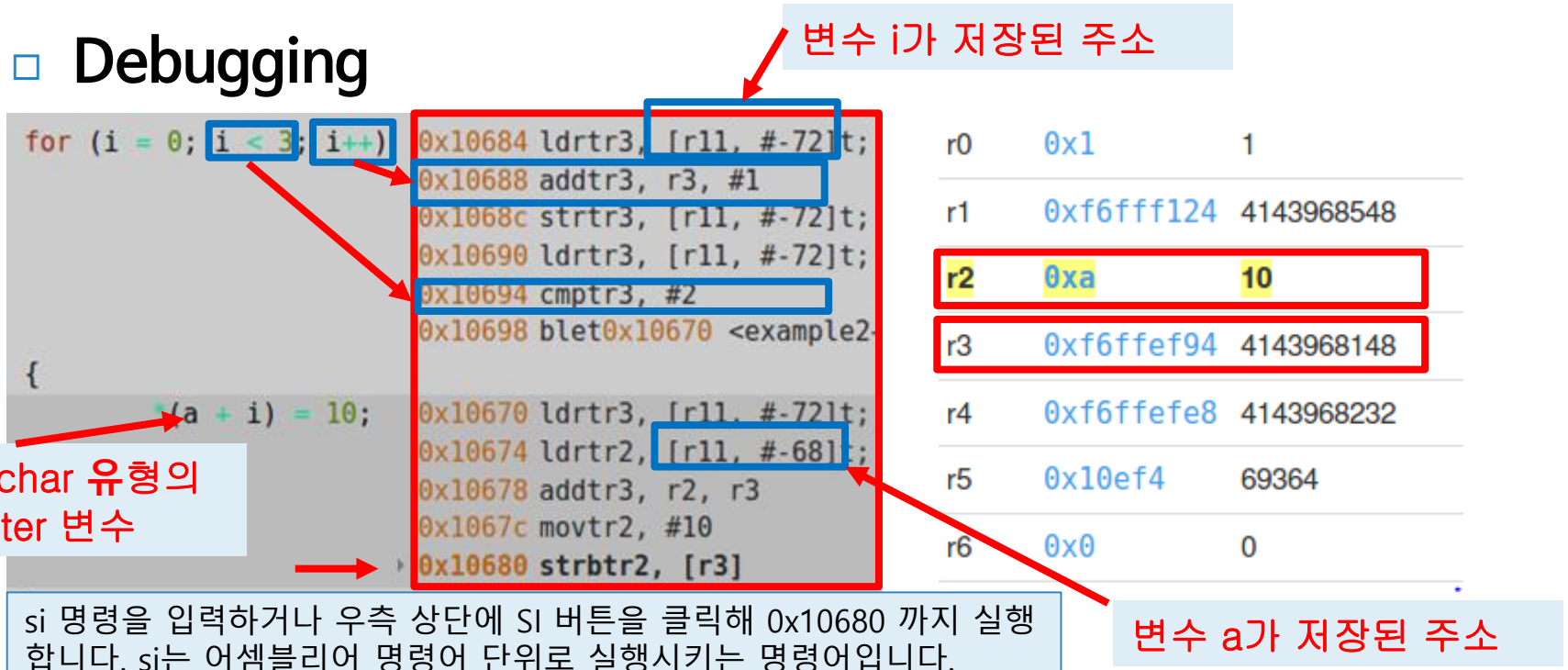
대표적으로 세 가지 크기가 다른 자료형을 가지고 실습을 해볼 거예요.

**만일 mem[40] 배열이 메모리 1000 번지부터 시작한다면 mem+4는 1004번지를, mem+16은 1016 번지를 의미함.**

for loop가 한번 수행될 때마다 멈추도록 break point를 설정한 후 disassembly section을 통해 확인해봅시다.

# Lab 2-2 : C pointer 동작 사례 2

- Debugging




si 명령을 입력하거나 우측 상단에 SI 버튼을 클릭해 0x10680 까지 실행합니다. si는 어셈블리어 명령어 단위로 실행시키는 명령어입니다.
**어셈블리를 해석하면 r2에 a의 주소, r3에 i값을 가져오고 매 for 루프마다 r3를 1만큼 증가시키고 r2+r3 메모리 주소에 10을 저장하고 있습니다. 다시 말하면, a의 주소에 +i만큼 더한 곳에 값을 저장한다는 의미와 같아요.**

즉, 0xf6ffef94 이라는 주소 (변수 a 저장주소 + i)에 10을 저장한다! 라는 뜻입니다.

그럼 메모리에 잘 들어갔는지 확인해봅시다.

# Lab 2-2 : C pointer 동작 사례 2

Network Computing Lab.

- Debugging

```
for (i = 0; i < 3; i++)
{
        *(a + i) = 10;
}
```

```
0x10684 ldrtr3, [r11, #-72]t;
0x10688 addtr3, r3, #1
0x1068c strtr3, [r11, #-72]t;
0x10690 ldrtr3, [r11, #-72]t;
0x10694 cmptr3, #2
0x10698 blet0x10670 <example2+
0x10670 ldrtr3, [r11, #-72]t;
0x10674 ldrtr2, [r11, #-68]t;
0x10678 addtr3, r2, r3
0x1067c movtr2, #10
0x10680 strbtr2, [r3]
```

memory

| 0xf6ffef94 | 0xf6ffefb3 | 4 |
|---|---|---|

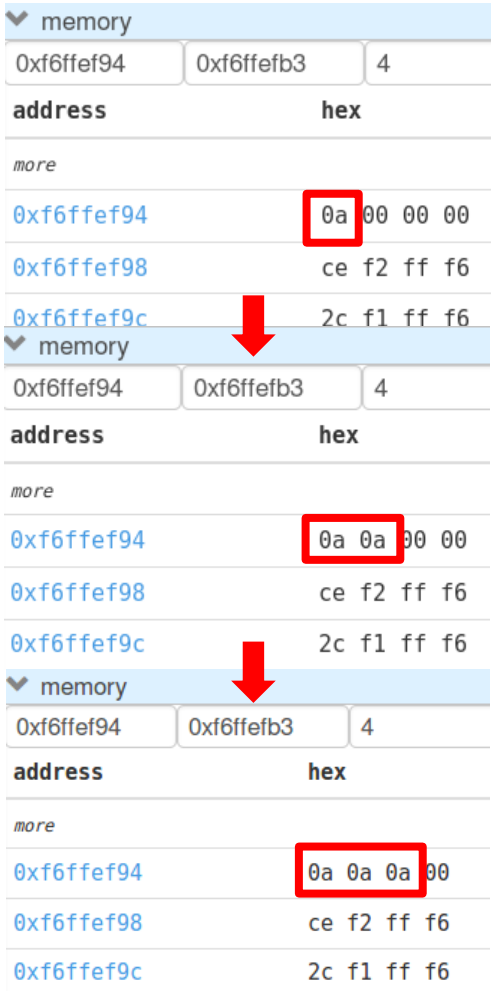
| address | hex |
|---|---|
| *more* | |
| 0xf6ffef94 | 0a 00 00 00 |
| 0xf6ffef98 | ce f2 ff f6 |
| 0xf6ffef9c | 2c f1 ff f6 |
| 0xf6ffefa0 | 00 00 00 00 |
| 0xf6ffefa4 | 18 99 04 00 |
| 0xf6ffefa8 | 90 0f 01 00 |
| 0xf6ffefac | 00 00 00 00 |
| 0xf6ffefb0 | 01 00 00 00 |

**for 루프가 한 번 돌때까지 SI 버튼을 클릭해 보세요.**
**그리고 r3에 저장된 주소값을 클릭해서 메모리를 확인 해봅시다.**
**0x0f6ffef94 에 0x0000000a(10)가 들어가 있네요.**

# Lab 2-2 : C pointer 동작 사례 2

- Debugging

```
for (i = 0; i < 3; i++)   0x10684 ldrtr3, [r11, #-72]t;
                          0x10688 addtr3, r3, #1
                          0x1068c strtr3, [r11, #-72]t;
                          0x10690 ldrtr3, [r11, #-72]t;
                          0x10694 cmptr3, #2
                          0x10698 blet0x10670 <example2-
{
        *(a + i) = 10;    0x10670 ldrtr3, [r11, #-72]t;
                          0x10674 ldrtr2, [r11, #-68]t;
                          0x10678 addtr3, r2, r3
                          0x1067c movtr2, #10
                          0x10680 strbtr2, [r3]
}
```




i < 3을 만족하는 동안은 위처럼 for loop를 순회할 거예요.

**계속해서 메모리를 char 타입으로 인덱싱해서 값을 저장하는 것을 확인할 수 있습니다.**
**이 때, 주소에 더해지는 r3의 값이 char타입 크기만큼 커지는 것이 이번 예제에서 중요한 포인트입니다.**

# Lab 2-2 : C pointer 동작 사례 2

- Debugging

```
40  void example2()
41  {
42          char mem[40] = {0, };
43          char *a;
44          int * b;
45          long long *c;
46          int i;
47
48          a = (char*)mem;
49          for (i = 0; i < 3; i++)
50          {
51                  *(a + i) = 10;
52          }
53
54          b = (int*)(mem + 4);
55          for (i = 0; i < 3; i++)
56          {
57                  *(b + i) = 1;;
58          }
59
60          c = (long long*)(mem + 16);
61          for (i = 0; i < 3; i++)
62          {
63                  *(c + i) = 12;
64          }
65  }
```

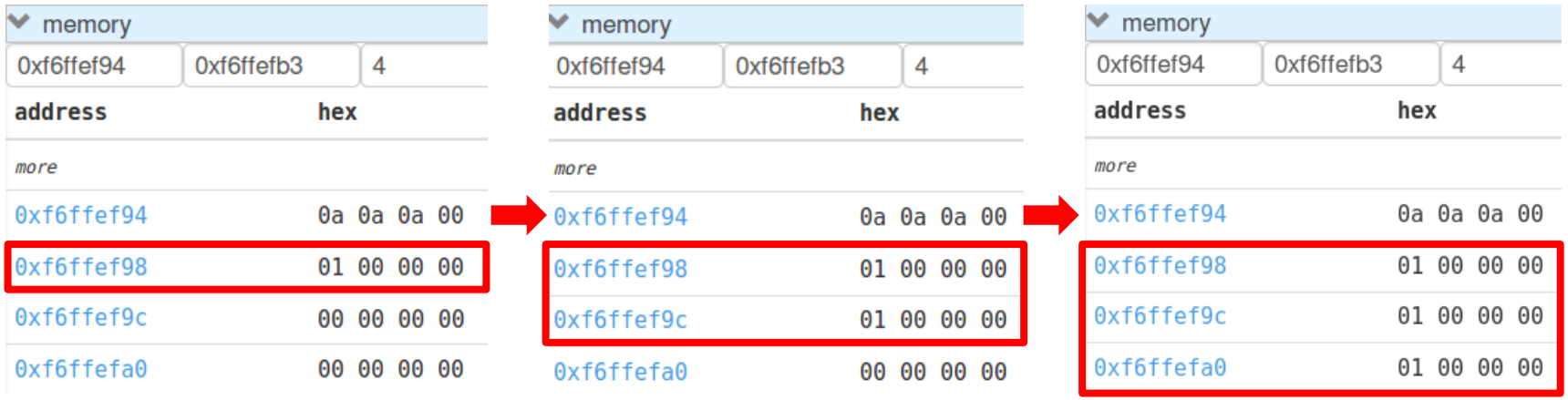
이제까지 char타입 변수가 메모리를 어떻게 접근하고 저장하는지 봤습니다.

**int타입, long long 타입도 어떻게 되는지** 과정은 각자 진행해 보시고, 결과가 되는 메모리를 보고 얘네들은 어떻게 인덱싱 하는지 확인해봅시다.

# Lab 2-2 : C pointer 동작 사례 2

- Debugging (int 형)

```
0x106b4 ldrtr3, [r11, #-72]t; 0xffffffb8
0x106b8 lsltr3, r3, #2
```




char타입과는 조금 다르게 메모리에 저장되는 것을 확인할 수 있죠?
char는 1byte 단위로 인덱싱했다면 **int는 4byte 단위로 인덱싱했네요.**

위에 assembly로 확인해보면,
**lsl r3, r3, #2는 r3에 left shift를 2회 수행하는 명령어입니다. 이 명령어 수행전 r3에는 i값이 저장됨. 이 명령어 수행후 r3에는 i * 4 한 값이 저장됩니다.**

# Lab 2-2 : C pointer 동작 사례 2

- Debugging (long long 형)

```
0x106fc ldrtr3, [r11, #-72]t; 0xffffffb8
0x10700 lsltr3, r3, #3
```

long long타입은 8byte 단위로 인덱싱 하는 것을 확인할 수 있습니다.

memory

| 0xf6ffef94 | 0xf6ffefcb | 8 |
|---|---|---|

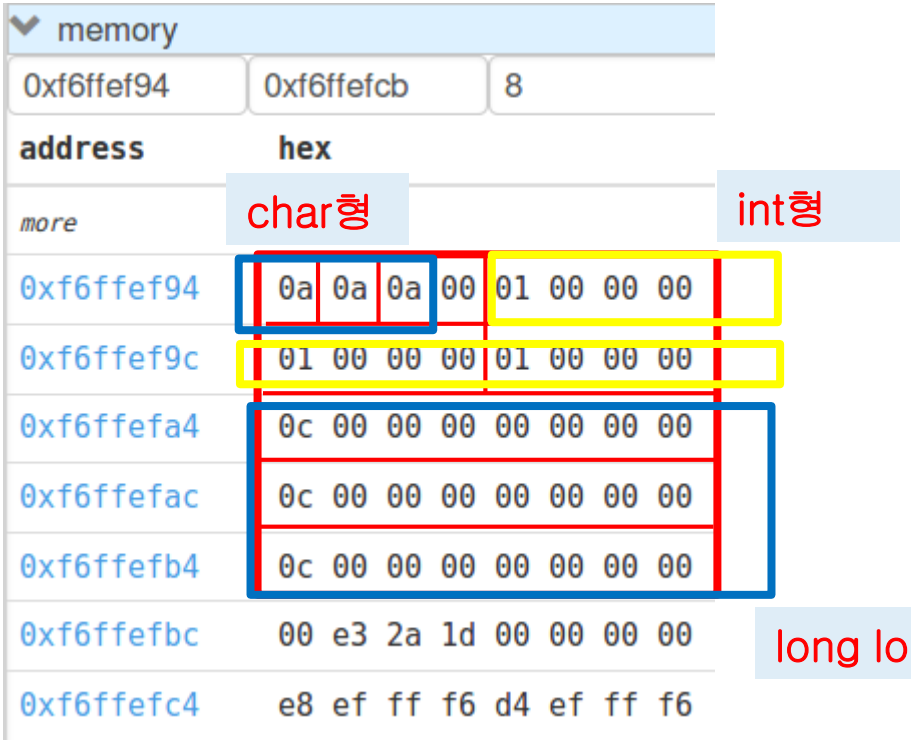
| address | hex |
|---|---|
| *more* | |
| 0xf6ffef94 | 0a 0a 0a 00 01 00 00 00 |
| 0xf6ffef9c | 01 00 00 00 01 00 00 00 |
| 0xf6ffefa4 | 0c 00 00 00 00 00 00 00 |
| 0xf6ffefac | 0c 00 00 00 00 00 00 00 |
| 0xf6ffefb4 | 0c 00 00 00 00 00 00 00 |
| 0xf6ffefbc | 00 e3 2a 1d 00 00 00 00 |
| 0xf6ffefc4 | e8 ef ff f6 d4 ef ff f6 |

# Lab 2-2 : C pointer 동작 사례 2

- Debugging

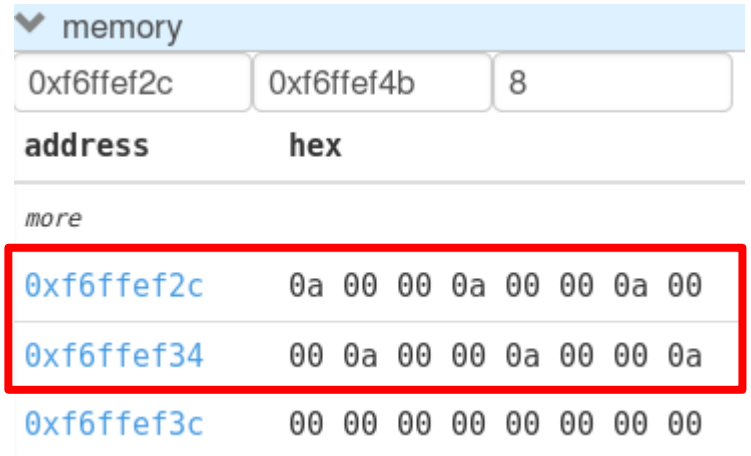
최종적으로 메모리 결과를 보면 오른쪽에 보이는 것처럼 값들이 들어가 있습니다.

char, int, long long 각기 변수의 데이터 크기 별로 메모리 인덱싱이 다른 것을 확인할 수 있었습니다.



# Lab 2-2 : C pointer 동작 사례 2

- **실습 B** : 자료형에 따라 메모리에 다른 크기로 접근하는 과정을 봤습니다. int형 배열을 선언 및 초기화하고 그 배열의 시작주소를 char형 포인터로 강제 캐스팅한 뒤, 그 포인터변수를 이용하여 배열의 내용이 아래와 같이 되도록 C 코드를 작성해보세요. 주석을 포함하는 코드와 결과 화면을 캡쳐하여 보고서에 첨부하세요.




* 메모리 주소 값은 다를 수 있습니다.

# Lab 2-3 : DT instruction indexing

- **DT(data transfer) 명령어는 ARM에서 기본적으로 str/ldr을 제공하고 있으며 주소 indexing 하는 방법이 여러가지 존재합니다. 이런 방법들을 익히면 더욱 최적화된 코드를 작성할 수 있을 것 입니다.**
  - Pre-indexing
  - Post-indexing
  - Auto-indexing

# Lab 2-3 : DT instruction indexing

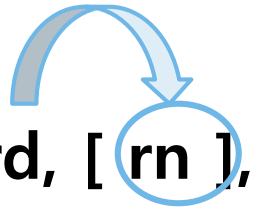
- **Pre-indexing**

```
str     rd, [ rn, +/-#constant ]
str     rd, [ rn, +/-rm ]
str     rd, [ rn, +/-rm, shift #constant ]
```

이런 식으로 pre-indexing 방식은 **offset을 먼저 계산한 후에** 해당 메모리에 값을 store하거나 load하는 방식입니다.

# Lab 2-3 : DT instruction indexing

- **Post-indexing**



```
str     rd, [ rn ], +/-#constant
str     rd, [ rn ], +/-rm
str     rd, [ rn ], +/-rm, shift #constant
```

반면, Post-indexing 방식은 pre-indexing과는 다르게
**먼저 Rn이 가리키는 메모리에 store/ load를 한 후에**
base register가 되는 **Rn의 주소를 업데이트 해줍니다.**

# Lab 2-3 : DT instruction indexing

- **Auto-indexing**

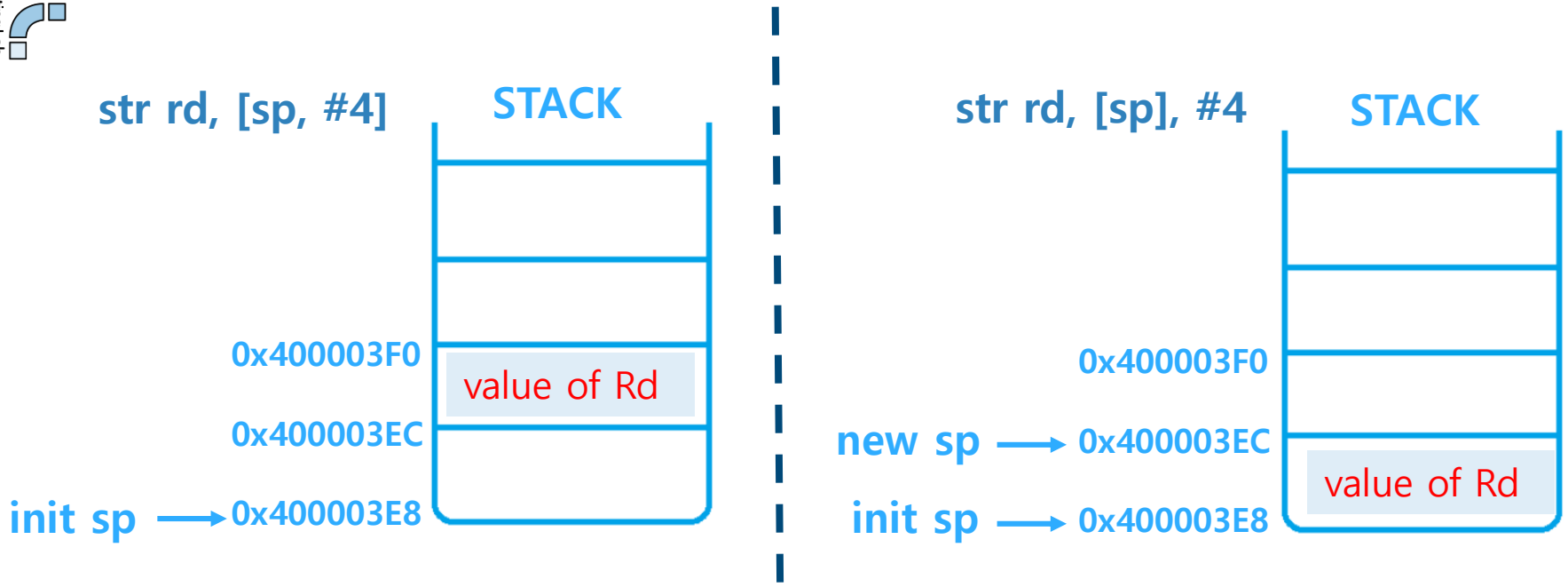
```
str     rd, [ rn, +/-#constant ] !
str     rd, [ rn, +/-rm ] !
str     rd, [ rn, +/-rm, shift #constant ] !
```

Auto-indexing 방식은 pre와 post를 섞었다고 보시면 돼요.
offset을 미리 계산하고 메모리에 저장하지만, 저장 후에
base register인 Rn의 값을 offset만큼 업데이트 해줍니다.

표현은 위와 같이 pre-indexing표현 뒤에 !를 붙이면 됩니다.

# Lab 2-3 : DT instruction indexing

- **Pre-indexing / Post-indexing**

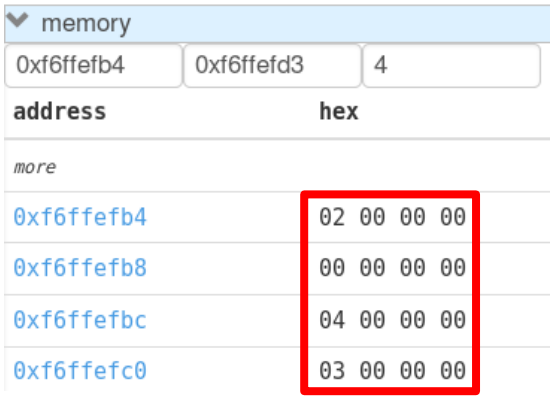



**str rd, [sp, #4]와 str rd, [sp], #4 를 비교한 그림입니다.**
초기 sp의 값은 같았다고 가정하고 같은 값을 stack에 저장한다고 가정해봅시다. 각자는 연산 후의 그림입니다.
data가 저장되는 주소가 다르고 sp의 업데이트 여부도 다르다는 것을 알 수 있습니다.

# Lab 2-3 : DT instruction indexing

- **실습 C** : 오른쪽 박스의 코드(example3 함수 다운로드)를 재빌드하세요.
- 실행 후 메모리를 확인하면 아래 화면을 볼 수 있습니다. 왜 이러한 결과가 나왔는지 indexing 관점에서 설명해 보세요. (13/16/22/26번줄 str 명령어 수행결과에 주목하세요). 이 내용을 보고서에 추가하세요.

※ 메모리 주소 값은 다를 수 있습니다.




```
1  .text
2  .global example3
3  .type example3, STT_FUNC
4
5  example3:
6    sub sp, sp, #12
7    str r3, [sp, #0]
8    str r4, [sp, #4]
9    str r5, [sp, #8]
10
11   sub sp, sp, #16
12   mov r3, #1
13   str r3, [sp, #0]
14
15   mov r4, #2
16   str r4, [sp], #4
17
18   mov r4, #0
19   str r4, [sp]
20
21   mov r5, #3
22   str r5, [sp, #8]!
23
24   mov r3, #4
25   mov r0, #-1
26   str r3, [sp, r0, lsl #2]
27
28   add sp, sp, #16
29   ldr r3, [sp, #0]
30   ldr r4, [sp, #4]
31   ldr r5, [sp, #8]
32
33   add sp, sp, #12
34
35   mov pc, lr
36
37   .end
```

# Lab 2-4 : C buffer overflow

- C 에서는 지역 변수 또는 지역 배열 변수들이 스택에 저장되는데 스택에는 임시 보존용 레지스터 값, 현재 수행함수 종료 후 리턴할 주소, 상태정보들이 함께 들어있습니다.
- C 코드에서 선언한 배열의 원래 크기보다 더 큰 인덱스를 부주의로 사용할 경우 심각한 문제 발생할 수 있습니다. C 컴파일러는 배열 참조 시 인덱스 범위를 체크하지 않습니다.
- 선언한 배열의 원래 크기보다 더 큰 인덱스를 부주의로 사용하여 메모리 쓰기 동작을 수행할 경우, 스택에 저장된 다른 값들(보존용 레지스터 값, 현재 수행함수 종료 후 리턴할 주소, 상태정보)이 깨지게 되며, 결과적으로 프로그램은 오동작을 하게 됩니다.

# Lab 2-4 : C buffer overflow

## Debugging

Break point

```
69  void example4()
70  {
71          int buf[5] = {0, };
72          int i = 0;
73
74          printf("\nInput buffer value: ");
75          while (i < 10)
76          {
77                  buf[i] = getchar() - '0';
78                  i++;
79          }
80
81          printf("\nPrint array: ");
82          for (i = 0; i < sizeof(buf)/sizeof(int); i++)
83          {
84                  printf("%d ", buf[i]);
85          }
86
87          printf("\n");
88  }
```

이 예제는 buf 배열의 크기를 5로 선언하고 나서 아래 while 문에서는 배열의 크기 5를 초과하여 입력을 받는 잘못된 코드입니다. buf 배열이 선언될 때 상태 정보를 저장하는 주소를 알기 위해 breakpoint를 설정합니다.

```
0x10788 movtr2, #0
0x1078c strtr2, [r3]
0x10790 strtr2, [r3, #4]
0x10794 strtr2, [r3, #8]
0x10798 strtr2, [r3, #12]
0x1079c strtr2, [r3, #16]
```

r2에 sp를 복사해서 사용하는 것으로 확인됩니다.

r3 0xf6ffefb0 4143968176

memory

0xf6ffefb0 | 0xf6ffefcf | 8

| address | hex | |
|---|---|---|
| *more* | | |
| 0xf6ffefb0 | 00 00 00 00 | 00 00 00 00 |
| 0xf6ffefb8 | 00 00 00 00 | 00 00 00 00 |
| 0xf6ffefc0 | 00 00 00 00 | 00 02 c0 81 |
| 0xf6ffefc8 | d4 ef ff f6 | c0 05 01 00 |

buf[0] buf[1]

buf 배열 외에 다른 데이터

스택의 아래 부분인 buf 배열(그림에선 위쪽에 0으로 초기화된 부분)이 확인되고 스택의 윗부분인 빨간색 박스 부분은 다른 코드들이 사용하는 데이터들이 저장된 것으로 보입니다.

# Lab 2-4 : C buffer overflow

- Debugging

```
69  void example4()
70  {
71          int buf[5] = {0, };
72          int i = 0;
73
74          printf("\nInput buffer value: ");
75          while (i < 10)
76          {
77                  buf[i] = getchar() - '0';
78                  i++;
79          }
80
81          printf("\nPrint array: ");
82          for (i = 0; i < sizeof(buf)/sizeof(int); i++)
83          {
84                  printf("%d ", buf[i]);
85          }
86
87          printf("\n");
88  }
```

example4() 함수에 진입해서 81, 88번 라인에 breakpoint를 설정하고 Continue를 하면 우측 사진 같이 입력을 받습니다.

```
youngjoon@ubuntu:~/CA_lab/lab2$ qemu-arm -g 8080 ./lab2

Input buffer value:
```

```
youngjoon@ubuntu:~/CA_lab/lab2$ qemu-arm -g 8080 ./lab2

Input buffer value: 1234567891
```

qemu를 실행시켰던 터미널에서 다음과 같이 10개의 숫자를 입력하고 엔터를 누릅니다.

```
youngjoon@ubuntu:~/CA_lab/lab2$ qemu-arm -g 8080 ./lab2

Input buffer value: 1234567891

Print array: 1 2 3 4 5
```

10개의 숫자를 입력했으면 다시 gdbgui로 돌아와 계속해서 Continue 합니다. 입력한 숫자들 중 buf 배열 영역에 속하는 5개의 숫자만 출력된 것을 볼 수 있습니다.

# Lab 2-4 : C buffer overflow

## Debugging

```
youngjoon@ubuntu:~/CA_lab/lab2$ qemu-arm -g 8080 ./lab2

Input buffer value: 123456789

Print array: 1 2 3 4 5
```

그냥 결과만 봤을 때는 아무 문제가 없는 것처럼 보여질 수 있습니다. 배열에는 원하는 대로 들어간 것 처럼 보이니까요.

그럼 이제 p28에서 확인던 buf 배열의 주소인 0xf6ffefb0 부근을 확인해봅시다.

memory

| 0xf6ffefb0 | 0xf6ffefe7 | 8 |
|---|---|---|

| address | hex |
|---|---|
| *more* | |
| 0xf6ffefb0 | 01 00 00 00 02 00 00 00 |
| 0xf6ffefb8 | 03 00 00 00 04 00 00 00 |
| 0xf6ffefc0 | 05 00 00 00 06 00 00 00 |
| 0xf6ffefc8 | 07 00 00 00 08 00 00 00 |
| 0xf6ffefd0 | 09 00 00 00 01 00 00 00 |
| 0xf6ffefd8 | 00 00 00 00 01 00 00 00 |
| 0xf6ffefe0 | 24 f1 ff f6 b4 05 01 00 |
| *more* | |

입력했던 문자 10개 1234567891의 값이 차례대로 잘 들어가 있는 것을 확인할 수 있습니다.

# Lab 2-4 : C buffer overflow

- Debugging

코드 수행후

memory

| 0xf6ffefb0 | 0xf6ffefe7 | 8 |
|---|---|---|

| address | hex |
|---|---|
| *more* | |
| 0xf6ffefb0 | 01 00 00 00 02 00 00 00 |
| 0xf6ffefb8 | 03 00 00 00 04 00 00 00 |
| 0xf6ffefc0 | 05 00 00 00 06 00 00 00 |
| 0xf6ffefc8 | 07 00 00 00 08 00 00 00 |
| 0xf6ffefd0 | 09 00 00 00 01 00 00 00 |
| 0xf6ffefd8 | 00 00 00 00 01 00 00 00 |

그래서 뭐요? 뭐가 문제라는 거죠? 라고 생각할 수 있어요. 자, 그래서 아까 처음 buf 선언하고 0으로 초기화 했을 때의 memory를 캡처해 놓은 것을 우측에 가져왔어요.

buf 초기화후

memory

| 0xf6ffefb0 | 0xf6ffefcf | 8 |
|---|---|---|

| address | hex |
|---|---|
| *more* | |
| 0xf6ffefb0 | 00 00 00 00 00 00 00 00 |
| 0xf6ffefb8 | 00 00 00 00 00 00 00 00 |
| 0xf6ffefc0 | 00 00 00 00 00 02 c0 81 |
| 0xf6ffefc8 | d4 ef ff f6 c0 05 01 00 |

선언 당시에는 배열의 원소의 개수가 5개라고 했기 때문에 5개의 공간만 주고 그 다음 스택에는 다른 어떤 값이 들어가 있습니다.

코드 수행후 파란색 영역 안 데이터가 깨짐

# Lab 2-4 : C buffer overflow

- Debugging

코드 수행후

memory

| 0xf6ffefb0 | 0xf6ffefe7 | 8 |
|---|---|---|

| address | hex |
|---|---|
| *more* | |
| 0xf6ffefb0 | 01 00 00 00 02 00 00 00 |
| 0xf6ffefb8 | 03 00 00 00 04 00 00 00 |
| 0xf6ffefc0 | 05 00 00 00 06 00 00 00 |
| 0xf6ffefc8 | 07 00 00 00 08 00 00 00 |
| 0xf6ffefd0 | 09 00 00 00 01 00 00 00 |
| 0xf6ffefd8 | 00 00 00 00 01 00 00 00 |
| 0xf6ffefe0 | 24 f1 ff f6 b4 05 01 00 |
| *more* | |

buf 초기화후

memory

| 0xf6ffefb0 | 0xf6ffefcf | 8 |
|---|---|---|

| address | hex |
|---|---|
| *more* | |
| 0xf6ffefb0 | 00 00 00 00 00 00 00 00 |
| 0xf6ffefb8 | 00 00 00 00 00 00 00 00 |
| 0xf6ffefc0 | 00 00 00 00 00 02 c0 81 |
| 0xf6ffefc8 | d4 ef ff f6 c0 05 01 00 |

우측 사진의 빨간색 박스를 보면 배열 바로 뒤에 값들이 스택에 들어가 있습니다. 어떤 정보인지는 알 수 없으나, 함수의 리턴 주소나 기타 중요하게 기억되어야 할 상태정보일 수 있다는 겁니다.
배열의 범위 이상으로 원소를 적재하여 overflow가 발생했습니다.
overflow로 인해 정보를 잃어 프로그램이 심각한 결과를 초래할 수 있다는 점이 이번 예제의 포인트입니다.

# Lab 2-4 : C buffer overflow

- 실습 D : main 함수에서 example4() 함수를 call 하자마자 스택에 저장하는 리턴 주소가 몇 번지 주소에 저장되는지 그 메모리 주소값, 그 주소에 저장된 리턴값을 디버거를 이용하여 확인하세요. (main 함수에서 example4() line에 breakpoint 를 걸어서 disassembly 화면을 보면 [address1] bl [address2] <example4> 코드가 보일 겁니다. address1 값이 bl 명령어가 들어 있는 메모리 주소입니다. bl 명령어 수행 후 return 해야할 주소는 [address1] + 4 입니다. bl 명령어를 수행하면 lr에 그 return 주소가 저장되고 함수 처음에 lr을 스택에 push할 겁니다. sp가 가리키는 메모리 근처에서 찾아보세요.)
- i값을 계속 증가시켜보면서 i값이 얼마일 때 위의 스택에 저장된 리턴값이 깨지는 지를 확인합니다. (스택의 최하위 주소를 가리키는 sp를 찾아 메모리를 주시하세요. 위에서 설명한 return 주소 lr이 메모리에서 i값으로 바뀔 때가 깨지는 상황입니다. 메모리가 깨지기 전 세 번의 루프를 포함해 중간 결과를 캡처하세요.)
- p28의 example4() 함수 코드에서 while문 내 i=10를 5로 수정, main 함수에 example4() 함수를 추가하고 재빌드하세요.
- 정상 수행되는지 확인하세요. 결과화면 및 배열이 저장된 메모리 내용을 확인하세요. (disassembly section을 보면 중간에 sp의 값을 저장해서 스택에 접근할 때 사용하는 다른 레지스터가 있습니다. 그 레지스터가 가리키는 메모리 주소 근처에서 찾아보세요.)
- 위 순서대로 실행하면서 중간결과들을 화면 캡처하여 보고서에 추가하세요.