



파이썬 - HW1

임베디드스쿨2기

Lv2과정

2020. 04. 16

박태

1. 배열 (1)

1) 배열을 사용하는 이유

- int 형 변수가 1000개 필요하다면 ???
 - ↳ 일일이 int a, b, c, d, e, f, g, h, ... zzz 까지 적기도 힘들다.
 - ↳ 여러개의 변수를 한번에 선언한다.
- 배열 선언 방법
 1. 먼저 다발로 **활용할 데이터 타입**을 적는다.
 2. 변수명이 있듯이 **배열의 이름**을 적는다.
 3. **얼마만큼의 공간을** 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)

```
// 배열의 길이 구하기
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
```

$$\frac{1}{\Gamma(\alpha)} \int_0^t (t-s)^{\alpha-1} f(s) ds = \int_0^t \frac{(t-s)^{\alpha-1}}{\Gamma(\alpha)} f(s) ds \quad (2.1)$$

이 시제 후보도 :를 사용해선 안 됩니다. :를 사용해선 안 됩니다

M. 동경시나 서울특별시에서 기거하는

[illegible]

이비정자 : A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 아 이 비 정 바 뱀 오 그 배 가 헤 어 히스 키에 켜의 휘저;

이제, \mathbb{R}^n 상의 함수 $f: \mathbb{R}^n \rightarrow \mathbb{R}$ 에 대하여

이러한 **배경요인**은 **배경 환경**을 형성하는 **조건**이 아니다.

1. 배열 (4)

```
0x0000555555551b1 <+8>: sub $0x20,%rsp
=> 0x0000555555551b5 <+12>: mov %TS:0x28,%rax
0x0000555555551be <+21>: mov %rax,-0x8(%rbp)
0x0000555555551c2 <+25>: xor %eax,%eax
```

자 이번에는 sub 명령어를 실행 했습니다.

Sub 명령어는 뺄셈 명령 입니다.

Sub 0x20, rsp는 현재 rsp 에서 32바이트를 빼겠다는 의미 입니다. ($0x20 \rightarrow 0010\ 0000 : 2^5 = 32$ 바이트,

가상 주소 공간에서는 바이트 단위로

움직입니다!)

그림으로 나타내어 보면 아래와 같습니다.

```
-----
| 0x0(rbp) | 0x0x7ffffffdfa0 ← rbp
-----
|           | <← sub로 만든 공간이 지역 변수 공간으로 쓰는 메모리 공간
-----
|           | 0x0x7ffffffdf80 ← rsp
-----
```

└ 이런 구조 때문에 스택은 아래로 자란다고 한 것이다!!

```
(gdb) x $rbp
0x7ffffffdfa0: : 현재 스택의 최상위 rsp 80
(gdb) x $rsp
0x7ffffffdf80: : 현재 스택의 기준점 rbp a0
```

1. 배열 (5)

```

=> 0x000055555555101 <+8>:    sub    $0x20,%rsp
0x0000555555551b5 <+12>:    mov     %fs:0x28,%rax
0x0000555555551be <+21>:    mov     %rax,-0x8(%rbp)
0x0000555555551c2 <+25>:    xor     %eax,%eax
0x0000555555551c4 <+27>:    movl    $0x2,-0x14(%rbp)
0x0000555555551cb <+34>:    movl    $0x4,-0x10(%rbp)
0x0000555555551d2 <+41>:    movl    $0x7,-0xc(%rbp)
0x0000555555551d9 <+48>:    movl    $0x3,-0x18(%rbp)
0x0000555555551e0 <+55>:    mov     -0x18(%rbp),%eax
0x0000555555551e3 <+58>:    mov     %eax,%esi
0x0000555555551e5 <+60>:    lea     0xe18(%rip),%rdi    # 0x555555556004
0x0000555555551ec <+67>:    mov     $0x0,%eax

```

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

↳ C 원문의 배열 선언

- Mov %fs:0x28, %rax
 - ↳ 보안상의 코드를 rax에 배치
- mov %rax, -0x8(%rbp)
 - ↳ rax 값을 rbp-8 에 배치
- xor %eax, %eax
 - ↳ xor의 특징은 값이 같으면 0이 나오죠?
이런 특징을 이용해서 eax를 0으로 초기화 했다는 걸 알 수 있습니다.
- movl \$0x2, -0x14(%rbp)
 - ↳ 2의 값을 rbp-20에 배치
- movl \$0x4, -0x10(%rbp)
 - ↳ 4의 값을 rbp-16에 배치
- movl \$0x7, -0xc(%rbp)
 - ↳ 7의 값을 rbp-12에 배치
- movl \$0x3, -0x18(%rbp)
 - ↳ 3의 값을 rbp-24에 배치
- mov -0x18(%rbp), %eax
 - ↳ rbp-24의 값인 3을 eax에 복사
- mov %eax, %esi
 - ↳ eax값을 esi에 복사
- lea 0xe18(%rip), %rdi
 - ↳ 배열의 값을 rdi에 배치
- mov 0x0, %eax
 - ↳ 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
0x000055555555
(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)

```
=> 0x0000555555551f1 <+72>: callq 0x555555550b0 <printf@plt>
0x0000555555551f6 <+77>: lea 0xe15(%rip),%rdi # 0x555555556012
0x0000555555551fd <+84>: callq 0x55555555090 <puts@plt>
0x000055555555202 <+89>: movl $0x0, -0x1c(%rbp)
0x000055555555209 <+96>: jmp 0x5555555522b <main+130>
0x00005555555520b <+98>: mov -0x1c(%rbp),%eax
```

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

```
(gdb) x $rsp-8
0x7fffffffdf78: 0x5555551f6
```

- lea 0xe15(%rip), %rdi
↳ 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: 0x555555202
```

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

- jmp 0x55...22b , 22b 위치로 점프

```
(gdb) si
0x00005555555522b 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

	0x555555202(복귀)	0x7fffffffdf78 ← rsp-8

1. 배열 (7)

```
=> 0x00005555555522b <+130>: mov -0x1c(%rbp),%eax
0x00005555555522e <+133>: cmp -0x18(%rbp),%eax
--Type <RET> for more, q to quit, c to continue without paging--
0x000055555555231 <+136>: jl 0x5555555520b <main+98>
0x000055555555233 <+138>: mov $0xa,%edi
0x000055555555238 <+143>: callq 0x55555555080 <putchar@plt>
0x00005555555523d <+148>: mov $0x0,%eax
0x000055555555242 <+153>: mov -0x8(%rbp),%rdx
0x000055555555246 <+157>: xor %fs:0x28,%rdx
```

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

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

```
=> 0x00005555555520b <+98>: mov -0x1c(%rbp),%eax
0x00005555555520e <+101>: cltq
0x000055555555210 <+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)

```
=> 0x00005555555520b <+98>:  mov    -0x1c(%rbp),%eax
0x00005555555520e <+101>:  cltq
0x000055555555210 <+103>:  mov    -0x14(%rbp,%rax,4),%eax
0x000055555555214 <+107>:  mov    %eax,%esi
0x000055555555216 <+109>:  lea    0xdfa(%rip),%rdi      # 0x555555556017
0x00005555555521d <+116>:  mov    $0x0,%eax
0x000055555555222 <+121>:  callq  0x555555550b0 <printf@plt>
```

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

```
(gdb) x $eax
0x2:  Cannot access memory at address 0x2
```

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

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

- callq 0x555... pritr
└ rsp-8에 복귀주소를 저장 한 뒤 jmp

```
=> 0x000055555555227 <+126>:  addl    $0x1, -0x1c(%rbp)
0x00005555555522b <+130>:  mov    -0x1c(%rbp),%eax
0x00005555555522e <+133>:  cmp    -0x18(%rbp),%eax
--Type <RET> for more, q to quit, c to continue without paging--
0x000055555555231 <+136>:  jl     0x5555555520b <main+98>
0x000055555555233 <+138>:  mov    $0xa,%edi
0x000055555555238 <+143>:  callq  0x55555555080 <putchar@plt>
```

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

	0x0(rbp)	0x7ffffffdfa0 ← rbp

	%rax	0x7ffffffdf98 ← rbp-8

	7	0x7ffffffdf94 ← rbp-12

	4	0x7ffffffdf90 ← rbp-16

	2	0x7ffffffdf8c ← rbp-20

	3	0x7ffffffdf88 ← rbp-24

	0 +1	0x7ffffffdf84 ← rbp-28

		0x7ffffffdf80 ← rsp

	0x55555227(복귀)	0x7ffffffdf78 ← rsp-8

1. 배열 (9)

```
=> 0x000055555555227 <+126>: addl    $0x1,-0x1c(%rbp)
0x00005555555522b <+130>: mov     -0x1c(%rbp),%eax
0x00005555555522e <+133>: cmp     -0x18(%rbp),%eax
--Type <RET> for more, q to quit, c to continue without paging--
0x000055555555231 <+136>: jle     0x5555555520b <main+98>
0x000055555555233 <+138>: mov     $0xa,%edi
0x000055555555238 <+143>: callq   0x55555555080 <putchar@plt>
```

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

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

```
// arr = { 2, 4, 7 }
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

	0x555555227(복귀)	0x7fffffffdf78 ← rsp-8

1. 배열 (10)

```
0x000055555555231 <+136>:  jl      0x55555555520b <main+98>
=> 0x000055555555233 <+138>:  mov     $0xa,%edi
0x000055555555238 <+143>:  callq   0x555555555080 <putchar@plt>
0x00005555555523d <+148>:  mov     $0x0,%eax
0x000055555555242 <+153>:  mov     -0x8(%rbp),%rdx
0x000055555555246 <+157>:  xor     %fs:0x28,%rdx
0x00005555555524f <+166>:  je      0x555555555256 <main+173>
0x000055555555251 <+168>:  callq   0x5555555550a0 <__stack_chk_fail@plt>
0x000055555555256 <+173>:  leaveq
0x000055555555257 <+174>:  retq
```

↳ for 문의 비교문이 끝나고 나면 위와 같이 다음으로 넘어간다.

- mov \$0xa, %edi
↳ 0xa의 값을 edi 에 배치

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

- callq 명령어
↳ 복귀 주소를 rsp-8에 push 후 점프

- mov \$0x0, %eax
↳ eax 값을 0으로 초기화

- mov -0x8(%rbp), %rdx
↳ 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
```

```
0x00005555555524f <+166>:  je      0x555555555256 <main+173>
0x000055555555251 <+168>:  callq   0x5555555550a0 <__stack_chk_fail@plt>
=> 0x000055555555256 <+173>:  leaveq
0x000055555555257 <+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

```
#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 수행하게 됨)
            // 결국 아래의 printf를 실행하지 않고 스킵하게 됨
            continue;
        }

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

    return 0;
}
```

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.

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

2. Continue(2)

됨)
스킵하게 됨

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

```
#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_num**은 num 변수의 주소 값

****pp_num**은 p_num 포인터 변수의 주소값이다.

printf의 *p_num은 p_num 포인터 변수가 가르키는 주소의 값을 출력한다.

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