



C언어 - HW5

임베디드스쿨1기

Lv1과정

2020. 04. 17

이충재

1. break

Break: 반복문 내에서 break를 만나면 반복문 바깥으로 빠져나온다.

주의할점: break코드가 위치하는 그 반복문만 빠져나온다.

```
#include <stdio.h>

int main(void)
{
    int i, j;

    for(i = 1; i <= 10; i++)
    {
        for(j = 1; j <= 10; j++)
        {
            if((j % 3) == 0)
            {
                printf("error\n");
                break;
            }

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

    return 0;
}
```

왼쪽 그림은 break가 위치하는 반복문만을 빠져나오는 것을 보여준다.

j가 3의 배수이면 error의 문구와 함께 프로그램이 종료 될 것 같아 보이지만 내부 반복문만 빠져나오기 때문에 프로그램이 계속 진행된다.

결과

```
i = 1, j = 1
i = 1, j = 2
error
i = 2, j = 1
i = 2, j = 2
error
i = 3, j = 1
i = 3, j = 2
error
i = 4, j = 1
i = 4, j = 2
error
i = 5, j = 1
i = 5, j = 2
error
i = 6, j = 1
i = 6, j = 2
error
i = 7, j = 1
i = 7, j = 2
error
i = 8, j = 1
i = 8, j = 2
error
i = 9, j = 1
i = 9, j = 2
error
i = 10, j = 1
i = 10, j = 2
error
```

(중간 생략)

```
i = 9, j = 1
i = 9, j = 2
error
i = 10, j = 1
i = 10, j = 2
error
```

2. continue

Continue: 반복문 내에서 continue를 만나면 continue 아래의 코드는 실행하지 않고 바로 반복문의 증감부로 이동한다.

```
int main(void)
{
    int i;

    for (i = 1; i <= 10; i++)
    {
        if (!(i % 3))
        {
            continue;
        }

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

    return 0;
}
```

1부터 10까지 3의배수를 제외하고 출력한다.
3의배수에서는 printf 함수를 실행하지 않는다.

결과

```
i = 1
i = 2
i = 4
i = 5
i = 7
i = 8
i = 10
```

goto

프로그램에서 Goto를 만나면 즉시 지정한 라벨로 이동한다.

사용방식:

Goto 라벨;

코드1

코드2

라벨:

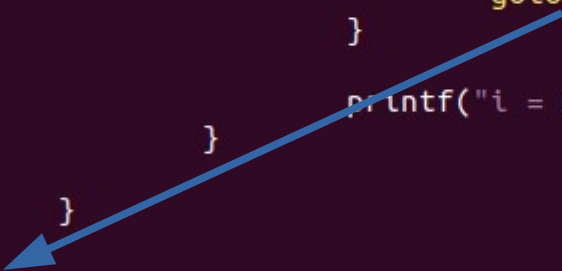
코드3

여기에서는 코드1, 코드2를 건너뛰고 코드3으로 이동한다.

```

int main(void)
{
    int i, j;
    for(i = 1; i <= 10; i++)
    {
        for(j = 1; j <= 10; j++)
        {
            if((j % 3) == 0)
            {
                printf("error\n");
                goto err;
            }
            printf("i = %d, j = %d\n", i, j);
        }
    }
err:
    printf("에러가 발생하였습니다.\n");
    return 0;
}

```



첫번째 페이지 break문 예시를 goto를 사용하여 수정한 것이다.

3의배수가 되면 바로 프로그램이 종료된다. 그 이유는 goto가 아래코드를 뛰어넘고 err로 이동하기 때문이다.

결과:

```

i = 1, j = 1
i = 1, j = 2
error
에러가 발생하였습니다.

```

이와 같이 goto는 여러개의 반복문을 빠져나올때 편하다.

- 배열 선언방법:
1. 자료형 배열이름[크기];
 2. 자료형 배열이름[크기] = {값, 값, 값};
 3. 자료형 배열이름[] = {값, 값, 값};

*주의점: 크기를 생략할때는 바로 배열의 요소를 결정해줘야 한다.
배열요소는 0부터 시작된다.

예시

```
int main(void)
{
    int name[5] = {1, 2, 3, 4, 5};

    printf("%d %d %d\n", name[0], name[2], name[4]);

    return 0;
}
```

Name[0] 는 배열의 첫번째 요소
name[2]는 세번째 요소
name[4]는 다섯번째 요소

배열명[n - 1] 은 n번째 요소

결과

1 3 5

배열의 전체공간 크기: sizeof(배열명)

배열 요소의 크기: sizeof(자료형)

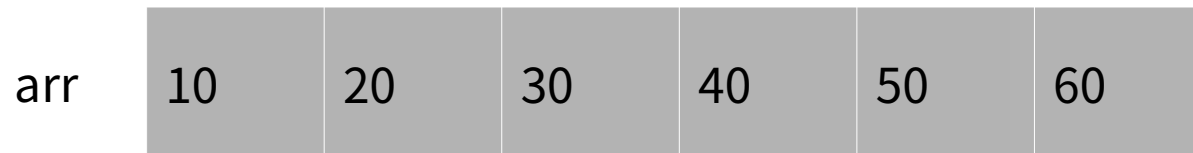
배열 요소 개수 구하기 :

$\text{sizeof(배열명)} / \text{sizeof(자료형)}$

요소의 개수는 전체크기를 요소한개의 크기로 나눈것과 같다.

ex) `int arr[6] = { 10, 20, 30, 40, 50, 60};`

`sizeof(int) = 4`



arr[0] arr[1] arr[2] arr[3] arr[4] arr[5]



`sizeof(arr) = 24`

이중배열 선언 방식: 자료형 배열이름 [첫번째크기][두번째크기];

예를들어 int num[3][3]은 [0][0], [0][1], [0][2]
[1][0], [1][1], [1][2]
[2][0], [2][1], [2][2]

총 9개의 요소를 가지고 있다.

예시)

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

    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 3; j++)
        {
            printf("%2d", arr[i][j]);

            if(j == 2)
            {
                printf("\n");
            }
        }
    }

    return 0;
}
```

크기가 3*3인 이중배열을 선언과 동시에
요소 값들을 설정해 주었다.

반복문을 통하여 요소를 출력하도록
프로그램을 작성하였다.

결과

1	2	3
4	5	6
7	8	9

포인터

포인터는 주소를 담는 메모리공간이다.

변수선언 방법: 자료형* 포인터이름;

```
int main(void)
{
    int num = 7;
    int* pointer = &num;

    printf("num = %d\n", num);
    printf("&num = %x\n", &num);
    printf("pointer = 0x%x\n", pointer);
    printf("*pointer = 0x%x\n", *pointer);

    return 0;
}
```

```
num = 7
&num = 8af6dcdc
pointer = 0x8af6dcdc
*pointer = 0x7
```

pointer에 num의 주소를 저장하였다.
따라서 &num과 pointer의 출력결과가 같다.

*포인터에서 *는 주소로의 접근을 의미한다.
따라서 *pointer는 pointer가 가지고있는
주소로 접근한다.

즉, *pointer = 0x8af6dcdc로 접근
= num으로 접근 => 0x7

이중포인터: 포인터의 주소를 담는 메모리공간

```
int main(void)
{
    int num = 3;
    int* pointer = &num;
    int**ppointer = &pointer;

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

    return 0;
}
```

ppointer 에 pointer의 주소를 저장
따라서 &pointer 주소값과
ppointer 값이 같다.

```
ppointer = 0x29f09968
&pointer = 0x29f09968
```

```
int main(void)
{
    int i;

    for (i = 1; i <= 10; i++)
    {
        if (!(i % 3))
        {
            continue;
        }

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

    return 0;
}
```

Continue 동작을 분석하기 위한 예시 코드이다.

```
0x000055555555149 <+0>:    endbr64
0x00005555555514d <+4>:    push    %rbp
0x00005555555514e <+5>:    mov     %rsp,%rbp
0x000055555555151 <+8>:    sub     $0x10,%rsp
0x000055555555155 <+12>:   movl    $0x1,-0x4(%rbp)
0x00005555555515c <+19>:   jmp     0x555555551a8 <main+95>
```

위의 코드로 인하여 rbp -0x4에 0x1이 저장된다.
그리고 0x~a8로 점프한다.

```
0x0000555555551a8 <+95>:   cmpl    $0xa,-0x4(%rbp)
0x0000555555551ac <+99>:   jle     0x5555555515e <main+21>
0x0000555555551ae <+101>:  mov     $0x0,%eax
```

그리고 0xa (10진수로 10) 과 rbp -0x4에 저장된 것(0x1)을 비교한다.
비교한 결과 저장된 값이 10보다 작으면 0x~5e로 점프한다.
크다면 eax를 0x0으로한다. 여기서 $10 > 1$ 이기 때문에 ~5e주소로 점프한다.

=> 이 과정은 for(i =1; i <= 10; i++)에서 i의 값을 비교하는 과정이다.

If(!(1 % 3))

```
0x00005555555515e <+21>: mov    -0x4(%rbp),%ecx
0x000055555555161 <+24>: movslq %ecx,%rax
0x000055555555164 <+27>: imul   $0x55555556,%rax,%rax
0x00005555555516b <+34>: shr    $0x20,%rax
0x00005555555516f <+38>: mov    %rax,%rdx
0x000055555555172 <+41>: mov    %ecx,%eax
0x000055555555174 <+43>: sar    $0x1f,%eax
0x000055555555177 <+46>: mov    %edx,%esi
0x000055555555179 <+48>: sub    %eax,%esi
0x00005555555517b <+50>: mov    %esi,%eax
0x00005555555517d <+52>: mov    %eax,%edx
0x00005555555517f <+54>: add    %edx,%edx
0x000055555555181 <+56>: add    %eax,%edx
0x000055555555183 <+58>: mov    %ecx,%eax
0x000055555555185 <+60>: sub    %edx,%eax
0x000055555555187 <+62>: test   %eax,%eax
0x000055555555189 <+64>: je     0x555555551a3 <main+90>
```

<+21>, <+24>: rbp -0x4에 저장된 값을 ecx로 옮기고 그것을 rax로 옮긴다.

<+27>: rax와 0x55555556을 곱한 값을 rax로 한다.

<+34>: rax를 오른쪽으로 32비트 쉬프트 연산을 한다. => 이 값이 나눗셈 몫 값이 된다.

<+38> ~ <+58>: 단순 값들을 더하고 옮기는 과정이다. 이 과정이 왜 일어나는지는 이해하지 못하였다.

<+60>: eax에서 edx를 뺀다. 뺄셈 결과가 나눗셈의 나머지가 된다.

<+62>, <+64>: eax를 and연산하여 1이면 아래 코드로 넘어가고 0이면 a3주소로 점프한다.

```
0x00005555555518b <+66>:  mov    -0x4(%rbp),%eax
0x00005555555518e <+69>:  mov    %eax,%esi
0x000055555555190 <+71>:  lea    0xe6d(%rip),%rdi    # 0x555555556004
0x000055555555197 <+78>:  mov    $0x0,%eax
0x00005555555519c <+83>:  callq  0x55555555050 <printf@plt>
0x0000555555551a1 <+88>:  jmp     0x555555551a4 <main+91>
0x0000555555551a3 <+90>:  nop
0x0000555555551a4 <+91>:  addl    $0x1,-0x4(%rbp)
0x0000555555551a8 <+95>:  cmpl    $0xa,-0x4(%rbp)
0x0000555555551ac <+99>:  jle     0x5555555515e <main+21>
0x0000555555551ae <+101>:  mov     $0x0,%eax
```

<+66> ~ <+83> : printf과정

eax가 1, 즉 나머지가 0이 아니라면 rbp - 0x4에 저장된 값을 edi로 하고
eax를 0으로 초기화한다. 그리고 edi 값을 출력한다.

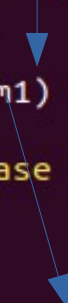
<+88> ~ <+101>: for(i = 0; i<=10; i++)과정

rbp - 0x4에 저장된 값을 1증가시킨다. 그리고 10과 비교하여 작으면 다시 ~5e주소로 점프한다.
10보다 크면 eax를 0으로하고 함수가 종료된다.

```
0x00005555555518b <+66>:  mov    -0x4(%rbp),%eax
0x00005555555518e <+69>:  mov    %eax,%esi
0x000055555555190 <+71>:  lea     0xe6d(%rip),%rdi    # 0x555555556004
0x000055555555197 <+78>:  mov     $0x0,%eax
0x00005555555519c <+83>:  callq   0x55555555050 <printf@plt>
0x0000555555551a1 <+88>:  jmp     0x555555551a4 <main+91>
0x0000555555551a3 <+90>:  nop
0x0000555555551a4 <+91>:  addl    $0x1,-0x4(%rbp)
0x0000555555551a8 <+95>:  cmpl    $0xa,-0x4(%rbp)
0x0000555555551ac <+99>:  jle     0x5555555515e <main+21>
0x0000555555551ae <+101>: mov     $0x0,%eax
```

만약 test 명령어를 수행한 결과가 $eax = 0$ 이었다면 0x~a3주소로 점프하기 때문에 출력을 건너뛰고 $rbp - 0x4$ 에 저장된 값을 1증가시킨다.

```
int main(void)
{
    int num1 = 2;
    int num2;
    switch(num1)
    {
        case 1: num2 = 4;
                break;
        case 2: num2 = 5;
                break;
        case 3: num2 = 6;
                break;
    }
    return 0;
}
```



num1의 값에 따라서 출력값이 달라지는 코드이다.

switch의 괄호 안의 값이 case 옆에 있는 값과 같은 곳을 찾아가 코드를 실행한다.

****주의점:** case의 끝에 break를 넣어주어야한다.
break를 넣지 않으면 아래코드도 실행된다.


```
0x000055555555129 <+0>:    endbr64
0x00005555555512d <+4>:    push    %rbp
0x00005555555512e <+5>:    mov     %rsp,%rbp
0x000055555555131 <+8>:    movl    $0x2,-0x8(%rbp)
```

rbp -0x8에 0x2를 저장한다.
이는 num1에 해당한다.

```
0x000055555555138 <+15>:    cmpl    $0x3,-0x8(%rbp)
0x00005555555513c <+19>:    je      0x55555555164 <main+59>
0x00005555555513e <+21>:    cmpl    $0x3,-0x8(%rbp)
0x000055555555142 <+25>:    jg      0x5555555516c <main+67>
0x000055555555144 <+27>:    cmpl    $0x1,-0x8(%rbp)
0x000055555555148 <+31>:    je      0x55555555152 <main+41>
0x00005555555514a <+33>:    cmpl    $0x2,-0x8(%rbp)
0x00005555555514e <+37>:    je      0x5555555515b <main+50>
0x000055555555150 <+39>:    jmp     0x5555555516c <main+67>
0x000055555555152 <+41>:    movl    $0x4,-0x4(%rbp)
0x000055555555159 <+48>:    jmp     0x5555555516c <main+67>
0x00005555555515b <+50>:    movl    $0x5,-0x4(%rbp)
0x000055555555162 <+57>:    jmp     0x5555555516c <main+67>
0x000055555555164 <+59>:    movl    $0x6,-0x4(%rbp)
```

<+15>, <+19>

3과 rbp -0x8에 저장된 것(0x2)을 비교
3과 같지 않기 때문에 아래 코드 실행

<+21>, <+25>

3과 rbp -0x8에 저장된 것(0x2)을 비교
3보다 크지 않기 때문에 아래 코드 실행

<+27>, <+31>

1과 rbp -0x8에 저장된 것(0x2)을 비교
1과 0x2가 같지 않기 때문에 아래 코드 실행

<+33>, <+37>

2와 rbp -0x8에 저장된 것(0x2)을 비교
위 두 값이 같기 때문에 0x~15b로 점프

<+50>

0x5를 rbp -0x4에 저장

```
int main(void)
{
    int i;
    int arr[] = { 2, 4, 7, 9 };

    return 0;
}
```

크기가 4이고 자료형이 int인 배열 Arr을 만들었다.

Arr배열에 2, 4, 7, 9를 할당하였다.

```
0x000055555555149 <+0>:    endbr64
0x00005555555514d <+4>:    push    %rbp
0x00005555555514e <+5>:    mov     %rsp,%rbp
0x000055555555151 <+8>:    sub     $0x20,%rsp
0x000055555555155 <+12>:   mov     %fs:0x28,%rax
0x00005555555515e <+21>:   mov     %rax,-0x8(%rbp)
0x000055555555162 <+25>:   xor     %eax,%eax
0x000055555555164 <+27>:   movl    $0x2,-0x20(%rbp)
0x00005555555516b <+34>:   movl    $0x4,-0x1c(%rbp)
0x000055555555172 <+41>:   movl    $0x7,-0x18(%rbp)
0x000055555555179 <+48>:   movl    $0x9,-0x14(%rbp)
0x000055555555180 <+55>:   mov     $0x0,%eax
```

<+27> ~ <+48>

Rbp - 0x20 에 0x2 <= arr[0]

Rbp - 0x1c에 0x4 <= arr[1]

Rbp - 0x18에 0x7 <= arr[2]

Rbp - 0x14에 0x9 저장 <= arr[3]

배열의 요소 값은 메모리에 연속적으로 저장된다.

```
int main(void)
{
    int arr[2][2] = {{1,2},{3,4}};

    return 0;
}
```

자료형이 int인 이중배열 arr를 만들었다.
arr에 1, 2, 3, 4를 할당하였다.

```
0x000055555555164 <+27>: movl    $0x1,-0x20(%rbp)
0x00005555555516b <+34>: movl    $0x2,-0x1c(%rbp)
0x000055555555172 <+41>: movl    $0x3,-0x18(%rbp)
0x000055555555179 <+48>: movl    $0x4,-0x14(%rbp)
```

<+27> ~ <+48>

rbp -0x20 에 0x1 <= arr[0][0]

rbp -0x1c 에 0x2 <= arr[0][1]

rbp -0x18 에 0x3 <= arr[1][0]

rbp -0x14 에 0x4 <= arr[1][1]

연속적으로 배열의 요소들이
저장되는 것을 볼 수 있다.

```
int main(void)
{
    int num = 7;
    int* pointer = &num;
    int** ppointer = &pointer;

    return 0;
}
```

변수 num에 7 저장
포인터 pointer에 num주소 저장
이중포인터 ppointer에 pointer주소 저장

```
0x000055555555164 <+27>: movl $0x7,-0x1c(%rbp)
0x00005555555516b <+34>: lea -0x1c(%rbp),%rax
0x00005555555516f <+38>: mov %rax,-0x18(%rbp)
0x000055555555173 <+42>: lea -0x18(%rbp),%rax
0x000055555555177 <+46>: mov %rax,-0x10(%rbp)
0x00005555555517b <+50>: mov $0x0,%eax
```

Rbp -0x1c에 0x7저장
>> num에 해당
Rbp -0x18에 rbp -0x1c주소 저장
>>pointer에 해당
Rbp -0x10에 rbp -0x18주소 저장
>> ppointer에 해당

포인터에 변수의 주소가 저장되고 이중포인터에 포인터 주소가 저장 된 것을 확인하였다.