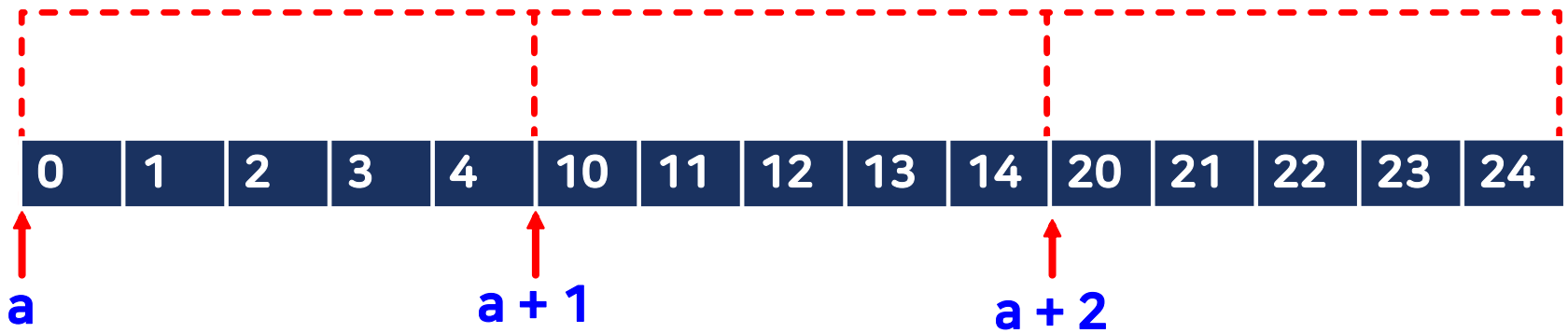
2차원 배열과 포인터

2차원 배열과 함수 (1/2)

```
int a[3][5] =
    { {0, 1, 2, 3, 4},
      {10, 11, 12, 13, 14},
      {20, 21, 22, 23, 24}
    };
```

0	1	2	3	4
10	11	12	13	14
20	21	22	23	24

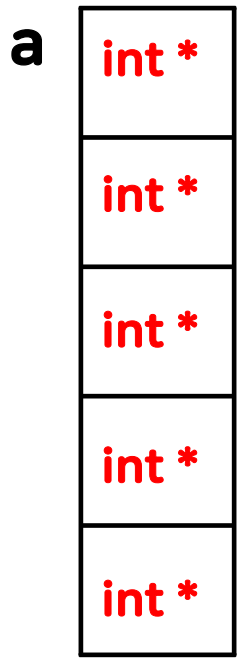


※ 위의 2차원 배열을 함수의 매개변수로
(1) void print_array(int a[][5]);
(2) void print_array(int (*a)[5]);

2차원 배열과 함수 (2/2)

- int *a[5];

int * 를 원소로 가지는 배열



0	1	2	3	4
10	11	12	13	14
20	21	22	23	24

- int (*pa)[5];

pa는 포인터라는 의미

int 형 변수를 의미

포인터 연산에 따른 증가 폭
즉, 여기에서는
sizeof(int) * 5의 크기만큼씩
pa + 1 에 해당하는 포인터
연산을 할 때, 증가한다는 의미

문자열 보충

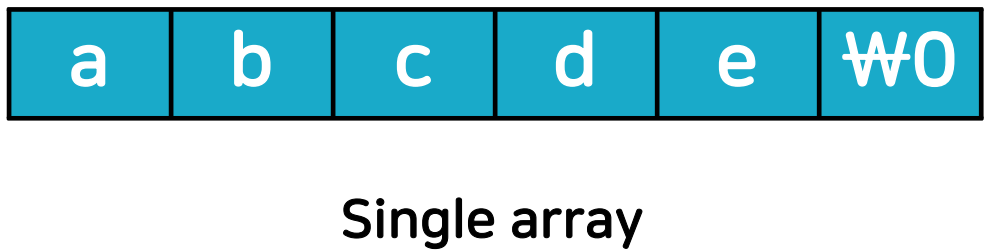
문자열 - 선언

- Arrays

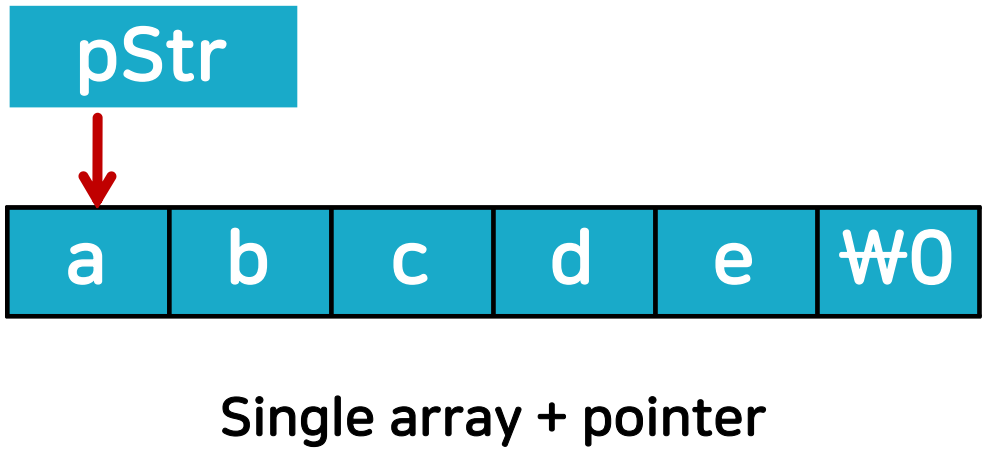
  char str[] = "abcde";

- Pointers

  char *pStr = "abcde";



Single array

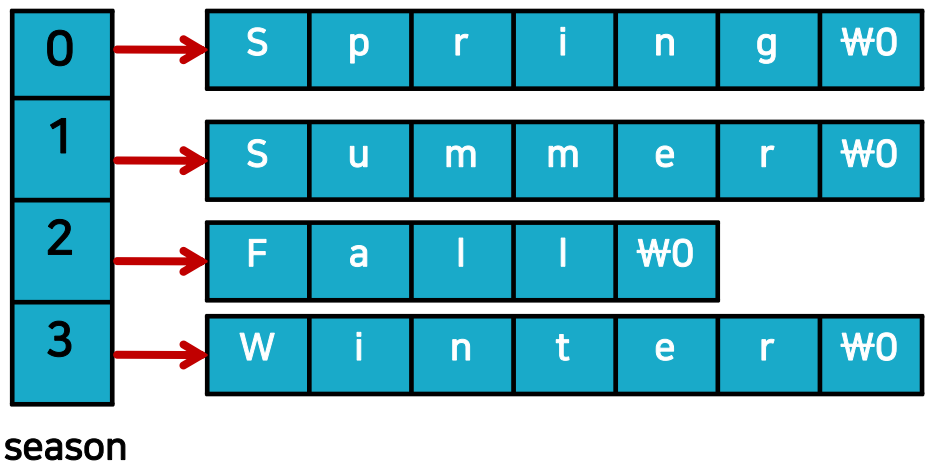


Single array + pointer

Arrays of String

- An array of character pointers is quite useful for working with an array of strings
  - char *season[4];
    - Create an array of four elements
    - Each element is a pointer to a character
    - Each pointer can be assigned to a pointer to string

```
char *season[4];
season[0] = "Spring";
season[1] = "Summer";
season[2] = "Fall";
season[3] = "Winter";
```



포인터 사용시 주의사항

포인터 이용시 주의사항

- 컴파일시 포인터 관련 warning은 절대 무시하지 말 것
- 초기화되지 않은 포인터는 절대로 사용하지 말 것
- malloc, calloc 등으로 메모리 할당을 했으면 반드시 free 함수로 할당을 해제해야 한다.

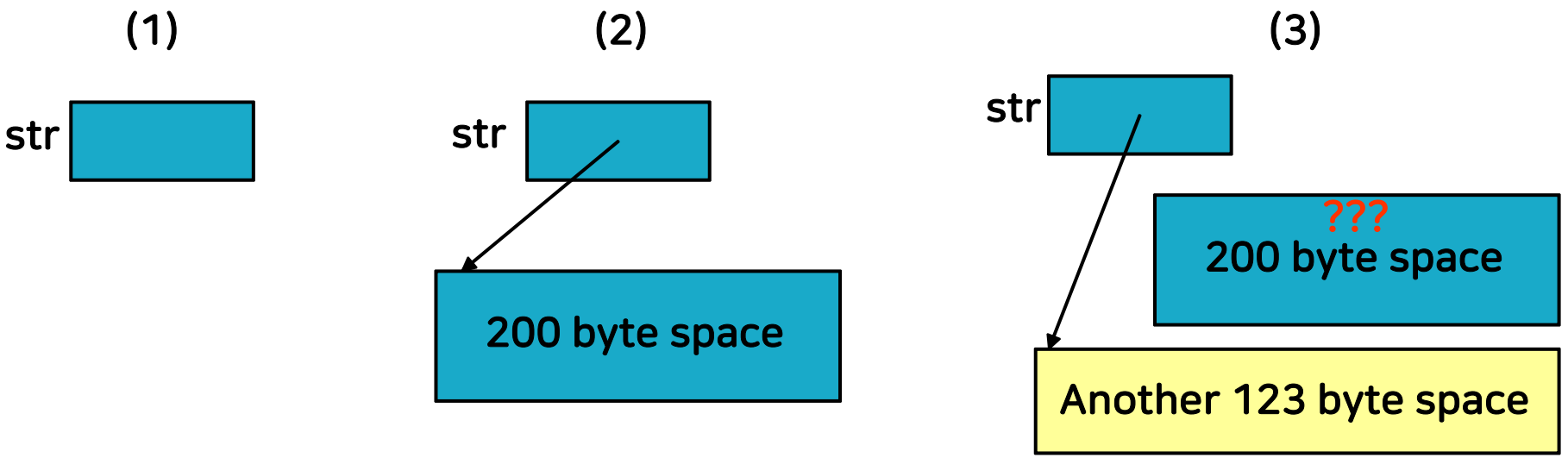
  참고) 배열이나 문자열처럼 컴파일러에 의해 자동 할당되는 공간은 알아서 할당 해제가 되니 걱정할 필요가 없다.
- 포인터 값이 합법적으로 할당된 메모리 공간의 주소 값인지 확인할 것 (그 위치를 읽을 때는 큰 상관이 없으나 그 위치에 값을 써 넣을 때는 특히나 주의…)

포인터의 부적절한 사용으로 인한 주요 부작용들

- 허공에 뜬 메모리 할당 공간 (이 공간을 가리키는 포인터가 존재하지 않는다)
- 길 잃은 포인터 (어느 위치인가를 가리키고 있기는 하지만, 정확하게 그 위치에 뭐가 있는지 알 수 없다.)
- 버퍼 오버플로우 (할당되어 있는 양 이상으로 메모리를 다루게 되는 경우)
- 잘못된 연산 오류 혹은 segmentation fault
- 컴퓨터의 오동작 (리부팅, 반응 없음, 기타 예측치 못한 현상들)

허공에 뜬 메모리 공간 (unreachable memory)

```
{
    char *str; (1)
    str = (char *) malloc(200 * sizeof(char)); (2)
... (no free() !!)
    str = (char *) malloc(123 * sizeof(char)); (3)
}
```

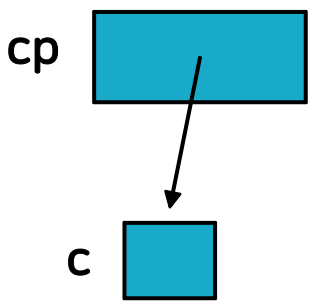


길 잃은 포인터 (dangling pointer)

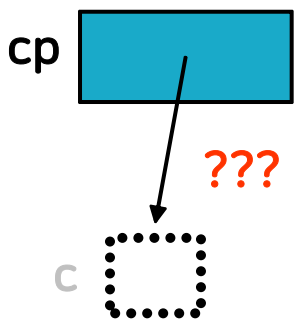
```
char *cp = NULL;  (1)
/* ... */
{
  char c;
  cp = &c; (2)
} /* c로 할당되었던 메모리 공간이 이 시점에서 할당 해제된다. */
/* cp는 이제 길 잃은 포인터(dangling pointer)가 되어버렸다. */ (3)
```

(1)

(2)



(3)



c 변수가 사라진 상황. 이후에 이 공간이 어떻게 사용될 지 아무도 모른다.

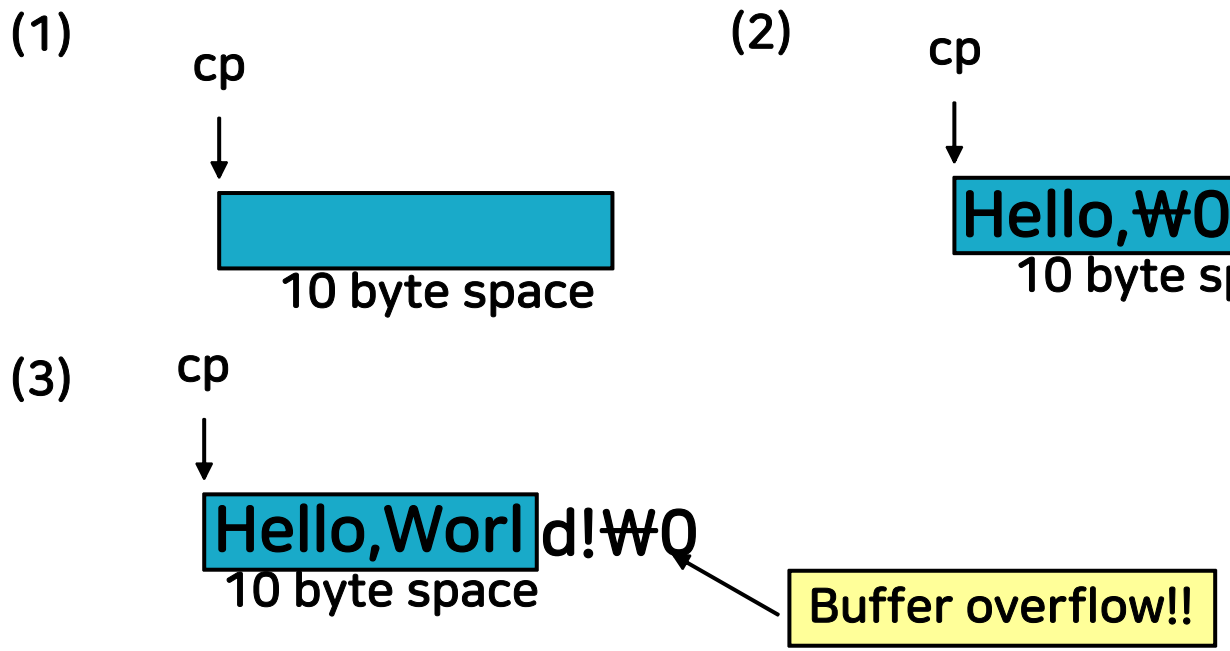
버퍼 오버플로우 (buffer overflow)

- 버퍼(buffer)
  - 대개는 임시로 어떤 내용을 저장하는 배열 형태의 저장소를 의미한다.
  - 컴퓨터 내에서 할당되어 사용되는 배열 형태의 임의의 메모리 공간이라고 봐도 무방하다.
  - 컴퓨터에서 일어나는 대부분의 작업은 효율성을 위해 버퍼를 많이 사용한다.

- 버퍼 오버플로우
  - 오버플로우란 넘친다는 의미이다.
  - 저장한 내용이 할당되어 있는 버퍼의 영역 바깥으로 넘치게 됨을 의미한다.
  - 만일 넘친 영역이 다른 용도로 사용되고 있었다면, 그 원래의 값을 파괴해버린다.
  - 넘친 영역의 값이 파괴됨을 이용하여 대상 컴퓨터를 혼란 시켜 침입하는 기법이 많이 사용된다.

버퍼 오버플로우의 예

```
char cp[10];  (1)
int a=0x12345678;
strcpy(cp, "Hello, "); (2)
printf("cp: %s a=%#x₩n", cp, a); → 의도한 대로 cp: Hello, a=0x12345678 이 출력됨
strcat(cp, "World!"); (3)
printf("cp:%s a=%#x₩n", cp, a); → 뭐가 출력될까? 잘못된 연산오류가 뜰지도 모른다.
        오류가 안뜬다면 문자열 넘침으로 인해 a의 값이 바뀌었음을 알 수 있다.
```



잘못된 연산 오류 (segmentation fault)

- 지정한 영역 이외에 엉뚱한 위치 (주로 null 혹은 쓰레기값)를 가리키는 포인터가 가리키는 위치에다가 어떤 값을 써 넣으려고 시도하는 경우에 발생한다.

- 현재 각 포인터가 가리키는 위치가 적법한지 꼼꼼히 살펴봐야 한다.

- 예제

  int *p; → 초기화 없이 선언

  *p = 123; → 초기화되지 않은 포인터가 가리키는 위치에 값을 쓰려고 시도한다.

  printf("%d₩n", *p); → 과연 실행될까?